A-E 400 648

Copy 43 of 113 copies

(12)

IDA PAPER P-1788

DoD RELATED SOFTWARE TECHNOLOGY REQUIREMENTS, PRACTICES, AND PROSPECTS FOR THE FUTURE

Samuel T. Redwine, Jr.
Louise Giovane Becker
Ann B. Marmor-Squires
R. J. Martin
Sarah H. Nash
William E. Riddle

June 1984

Prepared for

Office of the Under Secretary of Defense for Research and Engineering

IDA

INSTITUTE FOR DEFENSE ANALYSES

IDA Log No. HQ 84-28841

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. ADA145493	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DoD Related Software Technology Requirements, Practices, and Prospects for the Future		5. TYPE OF REPORT & PERIOD COVERED FINAL, Feb 84 - June 84
		6. PERFORMING ORG. REPORT NUMBER IDA Paper P-1788
7. AUTHOR(s) Samuel T. Redwine, Jr., Louise Giovane Becker, Ann B. Marmor-Squires, R.J. Martin Sarah H. Nash, William E. Riddle		8. CONTRACT OR GRANT NUMBER(s) MDA 903 84 C 0031
9. PERFORMING ORGANIZATION NAME AND ADDRESS Institute for Defense Analyses 1801 N. Beauregard Street Alexandria, VA 22311		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Task T-4-236
11. CONTROLLING OFFICE NAME AND ADDRESS STARS Joint Program Office 400 Army-Navy Drive, 9th Floor Arlington, VA 22202		12. REPORT DATE June 1984
		13. NUMBER OF PAGES 351
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) DoD-IDA Management Office 1801 N. Beauregard Street Alexandria, VA 22311		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE NA

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

computer programs; computer program reliability; computer program verification; military requirements; state of the art; computer programming; technology transfer; long range (time); military planning; software development

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This study investigates future DoD software requirements, current practices and approaches to software development, and the time it takes a software technology innovation to become widely used; and offers a glimpse of future possibilities in software technology.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

IDA PAPER P-1788

DoD RELATED SOFTWARE TECHNOLOGY REQUIREMENTS, PRACTICES, AND PROSPECTS FOR THE FUTURE

Samuel T. Redwine, Jr.
Louise Giovane Becker
Ann B. Marmor-Squires
R. J. Martin
Sarah H. Nash
William E. Riddle

June 1984

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

Contract MDA 903 84 C 0031
Task T-4-236

FOREWORD

This study is largely concerned with concrete examples and descriptions of DoD requirements and software technology. Except in drawing conclusions, we have tried to avoid the use of subjective expert judgement and high-level abstractions--not because these are bad but because the many prior studies in this area have generally followed that approach. Partially in the hope of communicating better, we have aimed at collecting and recording less subjective and less abstract descriptions of the areas covered.

The report is intended to supply data and background for those preparing material for Government decision makers, but should be of interest to all who have a significant interest in the Department of Defense (DoD) and software. It is concerned with assessing the state of the world and DoD plans; it does not try to formulate detailed solutions to the problems found. The report is concerned with the role of software in DoD's future plans and does not address larger issues such as the merit of these plans or non-DoD related issues such as the foreign efforts aimed at economic competition in computing and software.

The Executive Summary and the material in Chapters I, II, and VI --Introduction, Future Requirements, and Conclusions -- should be accessible to everyone. Chapter III on the current DoD-related software state of practice is reasonably accessible but requires some knowledge of software practices for full understanding. Chapter IV on the maturing of software technologies is also reasonably accessible but is enhanced if the reader has some knowledge of advanced software technology. Chapter V with examples of immature software technologies requires significant software knowledge for full appreciation.

ACKNOWLEDGMENTS

The authors are indebted to a number of people for their help on this study. Vance Mall, Carol Morgan, and Ken Nidiffer from the DoD; Betsy Bailey, Bill Carlson, and Tom Probert made several productive suggestions in the planning and review stages. Richard DeMillo and John Manley assisted in the technical review process and contributed case studies to Chapter IV. Other contributors of case studies to whom the authors are grateful are John Bailey, Paul C. Clements, and David Weiss. Assistance provided by Intermetrics, Inc. is also greatly appreciated. Special thanks go to Jo Ann Stilley, Charlene Smith, Mary Lou Caldwell, Anthea DeVaughan, Zelma Cameron, Joyce Tuggles, and Carol Powell for typing the manuscript. The authors wish to thank all others not mentioned here for their technical reviews.

RE: Classified References, Distribution Unlimited
No change in distribution statement per Ms. Betty Pringle, IDA

TABLE OF CONTENTS

Foreword iii
Acknowledgments iv
Executive Summary ix

Chapter I - Overview

A. FOCUS OF THE REPORT 1

B. IMPORTANCE TO DOD OF SOFTWARE 2

C. CONTENT OF THE REPORT 7

REFERENCES 8

Chapter II - A Review of Defense Software Requirements

A. INTRODUCTION 9

1. Objective and Purpose 9

2. Two Prior Studies 10

B. FINDINGS AND ANALYSIS 13

1. Software Development Component of DoD Programs 13

2. Defense Long Range Plans 15

a. Army Futures Concepts 18
b. Air Force 2000 28
c. Focus 21 32
d. AFSC Vanguard Planning Summary 34
e. Navy Command and Control Plan 37
f. DTST Ballistic Missile Defense (BMD) 39

C. SUMMARY 42

REFERENCES 44

Chapter III - Current State of Practice

A.	INTRODUCTION	47
B.	NATURE OF SOFTWARE DEVELOPMENT AND MAINTENANCE	48
C.	PROJECT CHARACTERIZATION	49
	1. Project Overviews	49
	2. Software Applications	50
	3. Magnitude of Software Development Efforts	50
	4. Software Development Life Cycles	53
D.	PROJECT MANAGEMENT	53
	1. Costs	54
	2. Staffing	56
	3. Milestones and Schedules	57
E.	PRE-SOFTWARE DEVELOPMENT	59
	1. Concept Definition	60
	2. Systems Requirements Definition	60
F.	SOFTWARE REQUIREMENTS DEFINITION	61
G.	DESIGN	63
H.	CODING	66
I.	INTEGRATION	68
J.	TESTING AND EVALUATION	69
K.	CONFIGURATION CONTROL	72
L.	SOFTWARE SUPPORT ENVIRONMENTS	74
	1. Hardware Support	74
	2. Post Development Software Support	75

M. SUMMARY 76

Chapter IV - Technology Case Studies

A. INTRODUCTION 81

B. TECHNOLOGIES INVESTIGATED 82

C. TECHNOLOGY MATURATION 84

D. GENERAL OBSERVATIONS 86

1. Major Technology Areas 92

2. Technology Concepts 92

3. Methodology Technology 93

4. Consolidated Technology 94

E. ACCELERATION OF TECHNOLOGY MATURATION 94

1. Critical Factors 95

2. Inhibitors 97

3. Facilitators 98

F. CONCLUSIONS 100

REFERENCES 104

Chapter V - State of the Art

A. INTRODUCTION 107

B. DISCUSSION 111

1. Methods 111

2. Testing Technology 113

3. Static Analysis Techniques 114

4. Verification Techniques 115

5. Program Transformers 117

6. Pre-Implementation Modelling Notations 118

7. Measurement Technology 120

8. Compiler Generation Technology 121

9. Software Engineering Environments 123

10. Editors 125

11. Command Languages 126

C. CONDUCT OF THIS SURVEY 127

REFERENCES 129

Chapter VI - Conclusions 135

APPENDICES

A. Methodology
B. Programs Surveyed-Categorized
C. Bibliography of Long Range Plans
D. DoD Long Range Planning Offices Contacted
E. Analysis of <u>Airland Battle 2000</u>
F. Navy C^2 Programmatic Action
G. Technology Case Studies

EXECUTIVE SUMMARY

Concern exists about a possible future software "gap" between the requirements for software imposed by DoD plans and systems and the state of practice's ability to meet those requirements. As an initial step toward gaining a concrete understanding of this gap, this report concentrates on four areas:

- Future software requirements in Defense systems--implications for software of some future DoD and Service plans for combat and technology.

- Current state of practice--in depth look at eight systems.

- Software technology transition--case histories of a number of software technologies and some generalizations from them.

- Current research state of the art--examples in a number of areas of what is in use and in the R&D pipeline (adding to prior state-of-the-art surveys).

After reviewing the evidence, the conclusion is reached that DoD could indeed suffer from such a substantial future gap unless steps are taken.

The importance of software in military systems is shown by a number of facts. The DoD Report to Congress on the FY 1985 Budget identifies 160 key programs. Of the programs identified in the report, at least 120 (or 75%) were determined to have a significant software component. And of the additional programs listed in the RDT&E Annex to the FY 85 DoD budget that have not yet begun, 80% have a significant software component.

These are just the largest programs and many others exist. The Air Force supplied a list of 238 Air Force programs with software development requirements, some of which overlap with the Secretary of Defense's report. In addition, Operation and Maintenance funds, not only Research and Development funds, pay for software efforts in the DoD.

The current development programs identified from the DoD budget report and elsewhere cover all types of systems from infantry close combat to strategic missiles. But as important and pervasive as software is to systems in existence or being developed today, its importance in future DoD systems appears to be even greater. Some idea of this increasing importance of software to these systems is demonstrated by such examples as

the X-29 Forward Swept Wing Aircraft and the proposed Ballistic Missile Defense System of the Strategic Defense Initiative -- both of which require software that represents advances over today's capabilities and that is critical to system operation. In addition, the recent experiences of the United Kingdom in the Falkland Islands and of the Israelis in Lebanon with software intensive weapons and systems show how software has influenced recent actual combat situations.

The X-29 supersonic aircraft with the forward swept wing design illustrates the critical nature of software to future systems performance. Totally dependent on software controlled electronic commands, the aircraft requires instant software controlled response and feedback to compensate for its instability as any pitch divergence can rapidly double in tenths of seconds causing its wings to break. These sorts of dependencies on software have led to estimates of needed future reliability levels for avionics software of a probability of failure of 10^{-9} over a ten hour period--a requirement many times greater than today.

Six DoD long-range planning documents were studied and analyzed for future technologies requiring software. Figure II-3, reproduced below, shows the percentage of items mentioned that require software in the five Service plans analyzed. The other plan analyzed was the DTST Ballistic Missile Defense Battle Management, Communications, and Data Processing plan that stated, "To design and create the software that ties this system together and makes it both effective and safe is probably the dominant problem for (DTST) battle management."

Many systems currently in the planning stages cannot operate without software, and the long range planning documents indicate that DoD system and mission performance will be increasingly dependent on software. Software reliability and its prediction are not well understood today and their importance will be rising as software grows in volume, criticality, and integrated functionality. It must be able to operate perfectly after prolonged periods of dormancy in hostile environments and when portions of the system have been destroyed. Altogether the picture painted by the study of future requirements is one of rapidly rising requirements along a number of dimensions.

A close look at a few major projects generally confirmed numerous prior reports on problems with the current state of practice meeting current requirements. Problems with budget, schedule, requirements, staffing, and product quality were found. Nevertheless, in these projects capabilities were being fielded and, while it was often a struggle, current requirements mostly are eventually being met.

SUMMARY OF FINDINGS - EXAMINATION OF SELECTED
SERVICE LONG RANGE PLANNING DOCUMENTS

Plan	Percentage requiring software*
Airland Battle 2000	74%
Air Force 2000	67%
Focus 21	67%
AFSC Vanguard Planning Summary	71%
Navy Command and Control Plan	73%

Source: Institute for Defense Analyses

*Minimum percentage of technologies, systems, functions, or actions enumerated in the plans.

Figure II-3

The study of fourteen software technologies indicates that bringing a technology to the point of maturation where it is popularized and disseminated to a large portion of the technical community generally has taken 15-20 years. Other studies indicate that 4-8 additional years may be required to propagate that technology throughout a large organization. Technological concepts and ideas that do not need computerized support or a change in the using community's mind set can mature faster. Particularly important to technology transition are a recognized need, a receptive target community, and a believable demonstration of cost/benefit. Well-designed channeling of attention and support, an articulate advocate, prior success, incentives, technically astute managers, readily available help, latent demand, simplicity, and incremental extensions to current technology were also identified as facilitators. Most significant among the technology transition inhibitors are the time it takes to transfer a technology internally, high cost, contracting disincentives, psychological hurdles, and the desire by programmers to "fiddle" with a technology that is too easily modified. Technology originators most often inhibited transition by small, simple mistakes that were easy to correct once identified. The Government has also not always facilitated technology transition.

The research state-of-the-art in software technology has been surveyed by a number of studies in recent years. An in depth look at eleven specific areas -- development methods, test technology, static analysis techniques, verification techniques, program transformers, modelling notations, measurement technology, compiler generation technology, software engineering environments, editors, and command languages -- confirmed through yet more examples the conclusions of prior studies that many potentially significant immature or unused technologies exist.

Seven conclusions were reached. The first five are:

Conclusion 1: Future DoD software requirements are rising rapidly and becoming increasingly critical to the DoD mission.

Conclusion 2: The current software state of practice is having difficulties meeting current DoD requirements.

Conclusion 3: Software technologies have taken significant time to reach widespread use -- 15 to 20 years.

Conclusion 4: The possibility exists to facilitate and accelerate software technology transition.

Conclusion 5: Many immature or unused software technologies exist that offer potential opportunities to improve the future state of practice.

The first three conclusions combined with consideration of two additional points lead to the conclusion that a potentially serious software problem exists. First, the state of practice relevant to meeting a requirement in a given year is the state over the several preceding years while the system was being specified, designed, and built. Second, requirements in the aggregate will present a significantly greater challenge than individually. As embedded software proliferates in non-DoD industries such as aviation, medicine, and communication, DoD could be faced with intense competition in the commercial marketplace for skilled human resources. Thus a substantial future gap between software related DoD requirements and the software state of practice is a real threat particularly given the criticality of software in many planned DoD systems.

Conclusion 6: A real potential exists for a critical future gap between DoD system and mission requirements and the future software state of practice's ability to meet them.

Conclusions 4 and 5, however, point to some opportunities that if properly exploited by DoD might help close this gap. Many immature technologies exist and their maturity and use could be accelerated. Problems in the current state of practice, however indicate that technology is not the only issue; management, acquisition, and personnel are also areas of concern.

Conclusion 7: Opportunities exist for DoD to help close the potential future software gap between requirements and the ability to meet them by accelerating technology transition combined with concern for management, acquisition, and human resources.

CHAPTER I OVERVIEW

A. FOCUS OF THE REPORT

This report was undertaken at the direction of the Department of Defense (DOD) to describe future DoD software requirements, examine current practices and approaches to software development, review the time it takes a software technology innovation to become widely used, and offer a glimpse of future possibilities in software technology. The term software denotes more than a collection of instructions to computer(s). It includes other descriptions such as requirements definitions, designs, and maintenance manuals as well as tests, plans, documentation, training materials, etc. The software state of practice in the Defense community producing DoD systems varies and appears to lag well behind the research state of the art in software. The process of transforming research state-of-the-art results into usable forms and actually getting them used is technology transition.

In addition to the lag between state of art and state of practice, another software "gap" of particular interest to DoD is the gap between the requirements for software imposed by DoD systems and the state of practice's ability to meet those requirements. As an initial step toward gaining a concrete understanding of this gap, this report concentrates on four areas:

- Future software requirements in defense systems--implications for software of some future DoD and Service plans for combat and technology.
- Current state of practice--in depth look at eight systems.
- Software technology transition--case histories of a number of software technologies and some generalizations from them.

o Current state of the art--examples in a number of areas of what is in use and in the R&D pipeline (adding to prior state-of-the-art surveys).

Of interest in this report are future DoD requirements and the potential future gaps between those requirements and the future states of practice -- where the future is roughly the FY 1988 through FY 2000. After reviewing the evidence, the conclusion is reached that DoD indeed could suffer from such a substantial future gap unless steps are taken.

B. IMPORTANCE TO DOD OF SOFTWARE

Computers and related technologies increasingly permeate current and proposed DoD modernization efforts. An essential ingredient to properly utilizing computer resources is the software component. Software controls the systems and potentially permits the flexibility required by DoD systems.

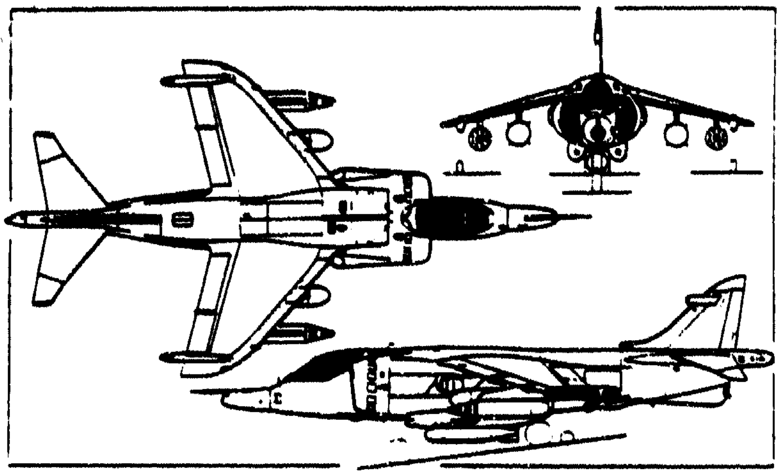
The importance of software in military systems is shown by a number of facts. The Report of the Secretary of Defense to the Congress on the FY 1985 Budget (1) identifies 160 key programs. This report contains brief descriptions of major DoD programs and, in most cases, development and procurement funding for fiscal years 1983-86. Of the programs identified in the report, at least 120 (or 75%) were determined to have a significant software component.

These are just the largest programs and many others exist. The Air Force supplied a list of 238 Air Force programs with software development requirements, some of which overlap with the Secretary of Defense's report. In addition, Operation and Maintenance funds, not only Research and Development funds, pay for software efforts in the DoD.

The current development programs identified from the DoD budget report and elsewhere cover all types of systems from infantry close combat to strategic missiles. But as important and pervasive as software is to systems in existence or being developed today, its importance in future DoD systems appears to be even greater. Some idea of this increasing importance of software to these systems is demonstrated by the examples of the X-29 Forward Swept Wing Aircraft, remote submersibles, and the proposed Ballistic Missile Defense System of the Strategic Defense Initiative -- all of which require software that represents advances over today's capabilities and that is critical to system operation. In addition, the recent experiences of the United Kingdom in the Falklands Islands and of the Israelis in Lebanon with software intensive weapons and systems show how software has influenced recent actual combat situations.

The X-29 supersonic aircraft with the forward swept wing design illustrates the critical nature of software to system performance. Totally dependent on software controlled electronic commands, the aircraft requires instant response and feedback at subsonic speeds to compensate for its instability at those speeds. (e.g., Any pitch divergence can rapidly double in tenths of seconds). Rapid and accurate data feedback is required to insure balance and reliable performance. In brief, while the software component is not necessarily the largest element of the system, it is vital to the functioning of the aircraft (2).

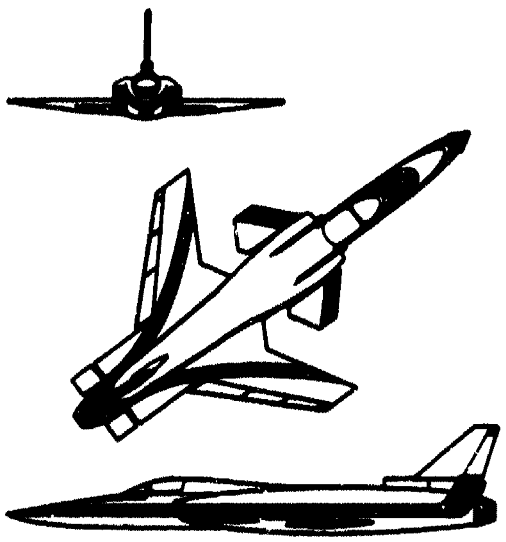
Another example of the critical nature of software is in the development and operation of remote autonomous submersibles. Remotely controlled underwater vehicles are being developed to replace humans with machines in dangerous underwater environments. Three types are under development: shark-type for search and inspection missions, crab-type for heavy-duty construction, rescue and repair, and octopus-type for repairing major structures underwater. All the component technology required for the shark-type is expected to be available



HARRIER GR. MK 3

Close Support and Reconnaissance Aircraft

Source: <u>Lessons and Implications from the South Atlantic Volume V Portfolio of Weapon Systems Used in Conflict,</u> Institute for Defense Analyses, November 1983. UNCLASSIFIED.



X-29 Forward Swept Wing Aircraft.
Photograph courtesy of
Grumman Aerospace Corporation.

commercially within five years. In ten years, crab-type vehicles will be available and it will take as long as fifteen years for the full development of octopus-type vehicles. Significant advances in software technology are essential to the development of submersibles. In addition, the importance of software reliability increases greatly as these vehicles become more autonomous, since a human will not be available to intervene (3).

The third example illustrates most vividly the complexity of major defense systems and the criticality of software in carrying out the system's mission: ballistic missile defense (BMD). As part of the DoD Strategic Defense Initiative, a Defensive Technologies Study Team was convened in 1983 to lay out a program of research and possible exploratory development focusing on problems of ballistic missile defense. The report of one of the panels, the Panel on Battle Management, Command, Control, and Communication and Data Processing describes the battle management system in terms of resources managed and functions performed. Major issues are discussed relating to the overall engineering and design of the system -- particularly the survivability of functions in the presence of battle damage, the criticality of the functions of ordnance safety and weapons release, and the rules of engagement that must support these functions. Reliability and communication requirements are addressed as well as the rates of data flow and computation for the large BMD system.

Issues of software design and development, one of the major engineering problems that the Panel forsees in the development of a BMD system, are also addressed. The following is excerpted from the introduction to the major conclusions of the Panel's report:

"Any BMD system will deal with tens of thousands of objects and probably several tiers of defense. The problem of achieving the computational speed and capacity needed to make decisions and to manage a complex and rapidly evolving battle has been emphasized in almost every study of antiballistic missile technology....It is the complex of broader engineering issues that dominates the Panel's concerns and recommendations. These issues relate to the difficulties of specifying and designing a system that will be of unprecedented complexity and to the reliability and safety of any resulting system that may finally be deployed.

A BMD system will be made up of many elements--sensors, weapons, computers, and data links, all controlled by complex software. Most of these will be replicated many times. Each by itself will be highly complex and will serve as one link in a figurative chain. All hardware and software links of this chain must function if the chain is to do its job. The software of the battle management system governs the coordinated activity of many chains, functioning side by side, and makes the whole aggregate of hardware behave as a purposeful entity. The engineering design of these elements and chains is a task that may be comparable in challenge and complexity to the Apollo program. To design and create the software that ties this system together and makes it both effective and safe is probably the dominant problem for battle management.

The problem is greater than just writing good software code, important as good code is. It is first a problem of systems design, bearing on the effectiveness, safety and economy of the BMD system as a whole, and then a problem of realizing that design, exactly, in reliable software (4)."

The problems confronted by the United Kingdom in the Falkland Islands conflict provide a recent example of the importance of software in military systems. For example, navigational systems, software dependent ordnance, and other equipment were identified as needing modification of the software to accommodate needed changes. Specifically, several critical systems' software was inadequate or failed and had to be modified to provide satisfactory performance (5). The GR-3 Harrier Aircraft weapon aiming computer software had to be modified to permit loft bombing (6). The navigation system of the Harrier required software changes to fly in the southern hemisphere (7).

Another software modification was to the Ferranti FE541 inertial navigation (FINRAE) system of the GR-3 to permit it to act as a reference system on a moving ship deck, giving the aircraft data for platform leveling and true north alignment before each sortie. Final software for the FINRAE was transmitted to the ships at sea via satellite communications. The British GR-3 pilots however, characterized this system as erratic and unsatisfactory even after the software modifications were made (8).

In contrast was the successful use, perhaps for the first time in support of actual operations, of computer graphics as an aid to the assessment of topography for the siting of an air defense system (9).

Another recent successful combat use of a software intensive system was the Israeli use of airborne surveillance and warning aircraft to manage their overwhelmingly successful air battle with Syria over Lebanon (10). Additional indication of the prowess of such technology is also given by the several recent U.S. deployments of the larger AWACS aircraft to middle-eastern trouble spots.

Although these examples are just highlights of defense systems, they are representative of the integral nature of software for managing and controlling essential functions in present and future DoD mission-critical systems.

C. CONTENT OF THE REPORT

This report consists of an analysis of future software requirements based on DoD planning documents and examples from DoD programs (Chapter II). The current state of practice is examined through a detailed look at a small number of systems (Chapter III). The time taken for software technology to progress from research to wide use, thereby improving the state of practice, is investigated via a number of software technology case studies (Chapter IV). Examples of what software technology is in the R&D pipeline are described (Chapter V). The report ends with a brief summary of findings and a discussion of their implications.

REFERENCES

(1) Report of the Secretary of Defense Casper W. Weinberger to the Congress on the FY 1985 Budget, FY 1986 Authorization Request and FY 1985-1989 Defense Program, Department of Defense, 1 February 1984. UNCLASSIFIED.

(2) Richard Demeis, "Forward Swept Wings and Supersonic Zip," High Technology, January 1982, p. 33. UNCLASSIFIED.

(3) John Douglas, "Remote Submersibles Take the Plunge," High Technology, Volume 3, Number 2, February 1983, pp. 16-17, UNCLASSIFIED.

(4) Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles--Volume V: Battle Management, Communications, and Data Processing. Brockway McMillan, Panel Chairman, Defensive Technologies Study Team, Alexandria, Virginia, October 1983, UNCLASSIFIED.

(5) Lessons and Implications from the South Atlantic Conflict, Volume II - Part 1: Appendices A-G - Background Papers, Institute for Defense Analyses, November 1983, SECRET.

(6) Ibid., p. F-4.

(7) Ibid., p. F-45.

(8) Ibid., p. F-3.

(9) Ibid., p. 0-11.

(10) Malcolm W. Brown, "Video Warfare Over Lebanon," Discover, Volume 2, Number 8, August 1982, UNCLASSIFIED.

CHAPTER II. A REVIEW OF DEFENSE SOFTWARE REQUIREMENTS

A. INTRODUCTION

1. Objective and Purpose

An appreciation of future DoD software requirements is essential to understanding the potential gap between expectations and the ability to meet Defense objectives. This chapter provides information on existing and projected software requirements in the Department of Defense.

The aim of this chapter is to identify software requirements integral to DoD operation and mission goals. Six selected DoD long-range plans were reviewed to identify approaches and specific technologies that infer a software component. In addition, a number of inquiries were made of selected DoD programs.

The software requirement, defined in this context, is an essential or critical element within a program or application. DoD software development often requires a large investment of resources and is essential to the mission or operation of the system or program.

This "criticality" is well illustrated by the X-29, the supersonic aircraft with the forward swept wing design discussed in Chapter I. The advantages inherent in forward swept wing design over aft swept wings--lower stall speed, reduced drag, higher maximum lift, improved distribution of internal fuselage volume, and better low speed maneuverability--make the basic design particularly well-suited to a wide variety of Navy and Air Force aviation requirements. Several aircraft designs have employed the configuration in the past, but development was stymied by structural limitations. It is structurally unstable and, at high speeds, forward swept wings tend to break off. These problems can now be solved with new composite material construction and computer software control.

In the X-29, the pilot operates the hydraulically powered aircraft control surfaces via digital electronic commands instead of a mechanical pushrod and crank linkage. Electronic commands and distributed microelectronics provide instant response and feedback, and save weight and space. Aerodynamically, the X-29 design is actually unstable in pitch at subsonic speeds. Any pitch divergence can rapidly double in a few tenths of a second. The divergence motion is sensed by a system of gyros and accelerometers monitored by computers. These computers continuously correct the pitch with small canard deflections, maintaining stable flight. Because the aircraft is moving at such high speeds and the time to diverge is so small, it would be impossible for a human to correct the divergence himself.

When the pilot maneuvers the X-29, a command deflects a control surface to rotate the aircraft. Since the X-29 is such a responsive unstable configuration, once it is rotated, it continues to rotate unless it is stopped by a reverse control command at the end of an intended maneuver. This is possible because software controls the X-29 by taking into account flight conditions (speed, altitude and acceleration), rotation rates, and engine conditions as well as pilot action.

2. Two Prior Studies

The Electronics Industries Association in 1980 prepared a ten-year forecast of software requirements for DoD data processing applications and embedded computer applications (1). The Defense embedded computer forecast addressed the U.S. military and aerospace market for militarized digital computers that are applied in real-time equipment operations to solve tactical, strategic and operational problems. The study estimated the annual cost of embedded computer software at almost $9 billion in 1984, and predicted that it could reach $32 billion by 1990 (see Figure II-1). In terms of the embedded computer resources devoted to software, in 1980 the software content was 65%, by 1985 the software content will increase to 80% and by 1990 software will account for 85% of the total embedded computer dollars (see Figure II-2).

An industry report published in June 1979 by Frost & Sullivan, Inc., presents an analysis and forecast of the U.S. military software market (2).

The DoD software requirements covered include tactical and strategic system software, embedded systems software, fault-tolerant software, distributed systems, real-time non-distributed software and simulation and modelling software. The application areas include ground-based, airborne, space-based and re-entry systems, shipboard command and decision, information processing, displays and graphics, sensors/detectors and countermeasures, communications as well as surveillance and intelligence.

The report concludes with an overview of the important development trends in the military software field and a prediction that military software will increase to 6% of the DoD annual budget in 1985. Thus, these two previous studies that have examined the DoD mission-critical system market, indicate that software is a critical aspect of a very significant number of these systems and the demand for software is steadily increasing.

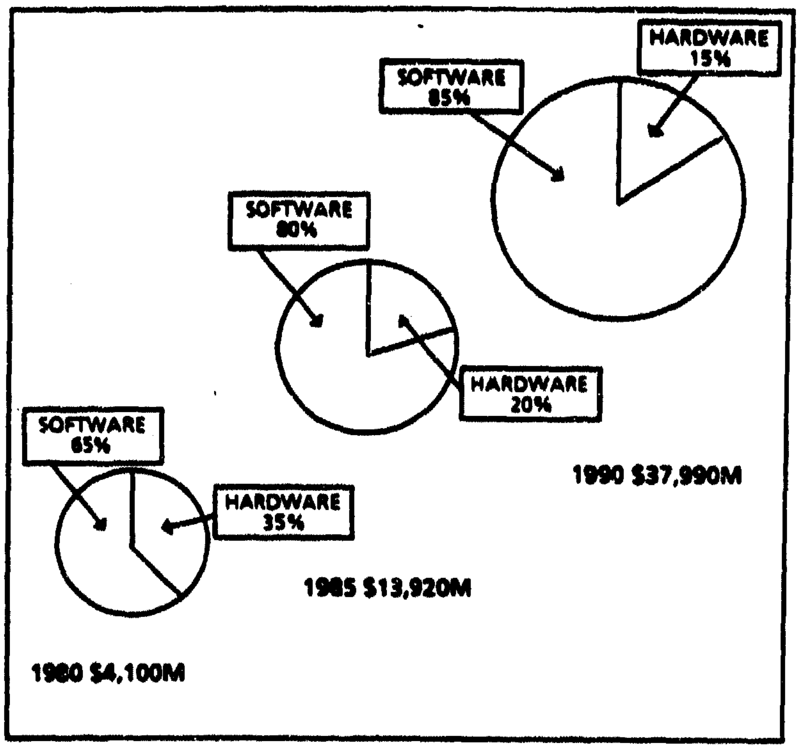


Figure II-1- Embedded Computers Hardware Vs. Software Cost

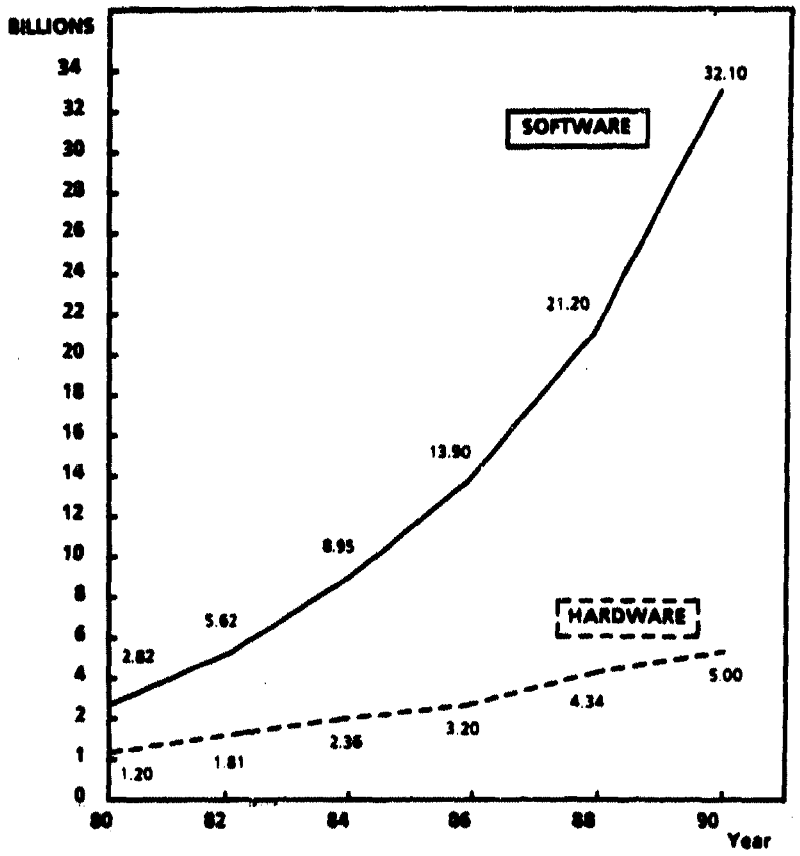


Figure II-2 - Embedded Computers Hardware Vs. Software Costs

Source: Electronic Industries Association

B. FINDINGS AND ANALYSIS

Information collected on major DoD programs, inquiries to DoD program managers about software development on their programs, and examinations of DoD long range planning documents reveal some findings about the status of software development on current projects and future requirements. These findings are analyzed for trends and future implications for DoD software development. Specifically, it is clear that current systems are unable to measure software reliability and that the volume, complexity, and criticality of future software requirements is increasing. Total expenditures for software development do not always accurately reflect the importance of software to a system or weapon. Therefore, future cost estimates for software development may actually understate the issue of criticality and not be good measures of future requirements.

1. Software Development Component of DoD Programs

Information collected on major current and future DoD programs revealed that (as discussed in Chapter I) at least 75% of the current programs have a software development component. Among the programs to begin in the 1985-1989 time frame, it was estimated that at least 80% will have a software development component. This suggests an increase in programs requiring software development for the near term.

Some specific findings relating to software development in major DoD programs follow. They concern software reliability and criticality of future software requirements. Software reliability is defined as the likelihood that the software performs as it was intended.

As systems become more autonomous, complex, and dependent on software control, software reliability gains importance. Yet, one of the most interesting findings was that most of the programs of whom inquiries were made were able to specify system reliability, usually mean time between failure (MTBF), but few were able to specify software reliability.

As an example of current efforts to address this problem, the Next Generation Weather Radar Program, a joint Department of Defense, Department of Commerce, and Department of Transportation program that uses twenty algorithms to process raw data from radar, described its software reliability requirements as follows. The program specifies three types of software errors (in the style of MIL-STD-1679 (3)): (1) an error that prevents the operating function from performing, (2) an error that degrades performance but can be worked around, and (3) all other intermittent errors that do not degrade performance. The measure of software reliability used by this program is that, upon completion of testing, the software should not have any known unresolved errors of type one, no more than one known error of type two per 70,000 machine instruction words and no more than one known error of type three per 35,000 machine instruction words. The survey respondent expressed doubts about the effectiveness of this measure of reliability but indicated it was the best measure currently available. (These doubts accurately reflect the current state of understanding of software reliability.)

Some systems that are highly dependent on software such as the LHX helicopter, currently under development by the Army, are creating triply redundant software for crucial systems such as flight control. The three systems will run in parallel, providing back up for one another in case of failure. Schemes such as this, while they require additional software development, at least recognize that software reliability is critical when the system depends heavily on software for safety and mission success. Very high future reliabilities will be required, for example the need has been estimated for the probability of failure in avionics of 10^{-9} per 10 hours flight operation (4).

Different functions require varying amounts of software. The WWMCCS (World Wide Military Command and Control System) Information System (WIS) and Joint Deployment System, both spend a high percentage of total program funds on software, as to be expected as C^3I (Command, Control, Communications, and Intelligence) systems make heavy use of computers. The A-6E Intruder aircraft however spent a very small percentage of total program funds on software since its costs are largely hardware-related. Nevertheless, though the software cost in this case is small, it is important to note that the software is critical to aircraft performance. The software enables the aircraft to navigate and to drop bombs in bad weather and at night by using radar to locate the target and sensors to tell the computer the speed of the airplane, the wind, and altitude.

2. Defense Long Range Plans

Six DoD long range planning documents were studied and analyzed for future technologies requiring software. In these plans, technology refers to a body of practical knowledge for achieving a purpose including scientific methods, tools, and procedures. These long range plans may or may not be formally approved by the Services, but they offer a glimpse of future technology directions.

- Airland Battle 2000
- Air Force 2000
- Focus 21 Appendix-Technological Opportunities for Focus 21
- AFSC Vanguard Planning Summary

- Navy Command and Control (C^2) Plan
- DTST Ballistic Missile Defense Battle Management, Communications, and Data Processing

Figure II-3 summarizes the findings, indicating the percentage of technologies, functions, and systems identified in the DoD service long range plans that require software*. On average, 70% of the technologies, functions, systems, and actions identified require software. Appendix A describes the methodology used to conduct these studies. Appendix C is a bibliography of long range planning documents including those that were examined for this study.

* The DTST Ballistic Missile Defense Battle Management, Communications, and Data Processing volume is not included in the figure as it did not identify specific technologies but nevertheless recognizes the importance of the software component and addresses means of improving software development.

SUMMARY OF FINDINGS - EXAMINATION OF SELECTED SERVICE LONG RANGE PLANNING DOCUMENTS

Plan	Percentage requiring software*
Airland Battle 2000	74%
Air Force 2000	67%
Focus 21	67%
AFSC Vanguard Planning Summary	71%
Navy Command and Control Plan	73%

Source: Institute for Defense Analyses

*Minimum percentage of technologies, systems, functions, or actions enumerated in the plans.

Figure II-3

a. Army Futures Concepts

The Airland Battle 2000 document describes an overall war fighting concept to drive organizational alignments, doctrine, training and materiel requirements for the Army in the early 21st century similar to the way that the Army Field Manual 100-5 Operations describes current airland battle doctrine. The Airland Battle 2000 examines trends in order to predict the environment, battlefield characteristics, and Service imperatives of the future. The Airland Battle 2000 and its successor, Army 21 (currently in the planning stages) describe technologies and systems of the future having sophisticated, massive, and critical software requirements.

The Airland Battle 2000 predicts that the battlefield of the 21st century will be dense with sophisticated combat systems for aerial and space surveillance, reconnaissance, target acquisition, air defense, and ultra fast communications. Systems will be integrated; there will be chemical, nuclear, biological, and electronic weapons; and robotics and artificial intelligence will play increasingly important roles (5). Integral to nearly all of these systems is software.

Some examples of systems currently in development or operation designed to meet the challenges of airland battle doctrine include sensor and fusion systems and remotely piloted vehicles. The JSTARS (Joint Surveillance and Target Attack Radar System) system being developed by the Army, will gather from air platforms and elsewhere in the battlefield information that will be fused and managed by a system such as the Joint Tactical Fusion System (also currently under development). In a combat situation, large amounts of unorganized information can be unusable, so decentralized, distributed, sophisticated information systems will need to be developed to organize, filter, and switch rapidly the information to appropriate destinations. In some cases, when time is of the essence, it may be appropriate to bypass the fusion system and downlink the information from the sensor directly to the artillery. Such sensors would require software to enable them to distinguish between information to be fused and information to be sent directly to a unit.

Airland Battle 2000 divides the future battlefield into nine functional areas. While they were developed in an unconstrained study, they do offer a view of possible technology applications for the future. To draw a picture of future software development needs, the nine appendices in Airland Battle 2000 were analyzed for technologies requiring software. The results appear in Figure II-4. Appendix E lists the requirements identified in each appendix that require software.

Altogether, software is necessary to at least 74 percent of the future technologies listed in the Airland Battle 2000. Even the Combat Service Support (CSS) functional area which has the lowest percentage will be "characterized by mobility, automation, and independent operations" in the 21st century according to the Airland Battle 2000 (6).

Seven out of the nine appendices in Airland Battle 2000 mention the use of robotics and artificial intelligence. Simulations will be used to train all levels of military personnel in the future. Automated land navigation systems will help the Army to conduct accurate, continuous combat when visibility is restricted (7). All assault vehicles will have automated mine detection, neutralization, and reporting devices.

Some striking examples of future capabilities described in Airland Battle 2000 requiring software include the following.

- Remotely deployed, intelligent, robotic, automatic weapon stations will move rapidly around the battlefield under software control providing countermobility capabilities.
- Terrain analysis systems will support obstacle planning, control, and reporting and analytical software will replace human terrain analysis at all levels (8).
- Software will enable fuses to identify friend and foe, and will be command, time, or time-extended detonated (9).
- Real time intelligence systems will continuously record, analyze, and report data about the enemy, weather, terrain, and obstacles on the battlefield. Topographical data bases will be updated by remote imagery scanning systems.
- Other data bases will contain information about friendly units, patterns, profiles, and high value targets making it possible to assess the situation from the enemy's point of view.

Software, as part of a weapons system, will control its functions: monitoring sensors, tuning transmitters, controlling vehicles, and dropping bombs, for example.

Software will also perform an organization and analysis function. Examples are data fusion, database management, and decision support systems that organize and make sense of many facts at once.

Software can be modified in response to changing requirements without changing or replacing the hardware. Software, therefore is becoming increasingly responsible for system flexibility and functionality.

Technologies - Airland Battle 2000

Summary of Software Requirements

Functional Areas	Technol., Functions, and Systems Requiring Software
Command and Control	88%
Close Combat	78%
Fire Support	62%
Concept for Air Defense	89%
Intelligence and Electronic Warfare	83%
Communications	100%
Combat Support, Engineer, and Mine Warfare	48%
Combat Service Support*	38%
Army Aviation*	82%

*Did not contain a "Focus on Technology" section. Technologies were derived directly from appendix body.

Source: Airland Battle 2000.

Figure II-4

Although the Airland Battle 2000 makes few statements about system or more specifically software reliability requirements, it does say that battles in the future will be continuous, fought during day, night, in all weather conditions and on all types of terrain, implying that both system and software reliability (the ability to withstand these rigorous conditions) are critical to success on the battlefield of the future. In describing command and control systems of the future, Airland Battle 2000 mandates "a system that precludes sudden and catastrophic interruption of air defense coverage (10)." In addition, air defense systems must be simple, durable, not manpower intensive, modular in design (to allow for product improvements without major system redesign), and maintain commonality with other Army and Service systems (11).

Software will perform functions previously performed by humans such as navigation and identifying approaching aircraft. This software will be modular, allowing modification of parts of it without replacing hardware or all of the software. If system failures occur, software should ensure that the system undergoes graceful degradation instead of catastrophic loss of operation.

Weapons systems performance in the future cannot accept significant degradation through the employment of countermeasures by the enemy; therefore, systems must incorporate multiple modes of operations (radio frequency, infrared, acoustic, optical, laser), all of which are under software control, to maintain tactical viability (12). To avoid degradation in capability in the event of communications loss, weapon systems must be optimized for autonomous (i.e., software controlled) operations. This requires each system's software controlled identification and target classification capability to be near perfect (13).

Future communications and data distribution systems which Airland Battle 2000 recognizes must be highly reliable, will achieve redundancy through relays and alternate routing of digital data among sensors, maneuver control, fire support, air defense, combat service support, and intelligence electronic warfare data (14). Since the battlefield of the future is projected to be dense with sophisticated systems, most of which will require software, success on the battlefield depends upon these systems (and their software). System and software reliability are therefore crucial issues for the future.

A few examples comparing today's capabilities with those called for in Airland Battle 2000 help clarify the enhanced requirements.

The LHX helicopter is an example of a future system called for by Airland Battle 2000 that is software intensive and in the planning stages today. The system will have a fully integrated/automated cockpit. It is planned to be a highly reliable, lightweight, and easily maintained weapons system. Technology maturing in the 1980s in composite structures, engines, drive systems, and avicnics will be the cornerstone of the LHX design (15).

The LHX will have a digital flight control system and a digital electronic fuel control system both of which are highly dependent on software and triply redundant. Triple redundancy means that three separate flight control systems each with independently developed software will operate in parallel because flight control reliability is so critical to aircraft survivability. In the event of a software failure in any of them, the aircraft will still have flight control.

The LHX will have a variety of surveillance and target acquisition sensors including radar, forward looking infrared sensors, and low light level television cameras mounted on the aircraft that can operate in low visibility situations. Software will fuse information from these sensors to an integrated general display. The LHX will also have weapons and fire control software. Simulation software will be used to influence the general design process.

In fact, the aircraft cannot be flown without software. It is the most complex helicopter ever to be developed and the level of sensor integration has never before been done on a helicopter. It is different from the existing Army AH-64 (Apache) and AHIP (Army Helicopter Improvement Program) helicopters in that it is planned to be a single pilot aircraft with multiple sensors integrated into a single display. The AH-64 and AHIP have FLIRS (Forward Looking Infrared Radar System) and low light level television sensors that drive individual displays. There is no fusion and little information integration capability. The AH-64 also has weapons and fire control systems. The AH-64 software consists of 190,000 lines of code and was essentially an integration of off-the-shelf software. The LHX, it is estimated, will require at least 500,000 lines of code most of which will be new. The LHX plans to use VHSIC technology and the Ada programming language (16). Full scale development for the LHX is planned for FY 1986 and the system is scheduled to be introduced into the field in the early 1990s (17).

In the area of Air Defense, smart or maneuvering projectiles will have software to guide them, report their orientation, and sort targets. Shoot-on-the-move, fire-and-forget, and self-initiating missiles will all require software to interpret return signals and images, to guide them, to adjust sights and trajectory, to set parameters, and to sort targets. Software in sensors and battlefield information systems, utilizing various modes of communication, including laser, particle beam, microwave, and non-nuclear EMP, will provide decision support information, apply logic, distribute or transfer assets, coordinate aircraft, report casualties, request replacements, and control inventories. Signature exploitation software will enable air defense systems to interpret return signals and images, recognize patterns, and sort targets. Early air defense systems, created after World War II, were capable of tracking only one target. The current Aegis system can track in excess of 200 targets and the Ballistic Missile Defense System is planned to track tens of thousands of targets.

As another example, Figure II-5 compares today's capabilities with future capabilities in the area of Air Defense, listing twice as many future capabilities as present capabilities, many of which will require software. The third column describes the functions of software in air defense.

Even Airland Battle 2000 does not cover all the future Army's software related requirements -- the Army is in the process of the next iteration of futures concepts development. The new concept will be called Army 21 and will subsume Airland Battle 2000, while also covering several new areas, including:

AIR DEFENSE CAPABILITIES

Today's Capabilities

Helicopters

Precision Guided Munitions

Tactical Ballistic Missiles

Cruise Missile

Aircraft

Strategic Surveillance Systems

Remotely Piloted Vehicles
- o Reconnaissance
- o Decoy
- o Destructive

Satellite

Airland Battle 2000 Capabilities

Rail gun (hypervelocity)

Smart or maneuvering projectiles

Directed Energy

Laser
Particle Beam
Microwave
Non Nuclear EMP

Countermeasures

Jammers
Obscurants
Aerosols

Missiles

Shoot-on-the-Move
Fire-and-Forget
Self-initiating Jammers

Antiair Mines and Barriers

Signature Exploitation

IR
UV
Acoustics
RF
Visible

Software Fuctions

Interpret return signals and images

Guide missile

Provide decision support information

Sense orientation

Adjust sights, trajectory

Pattern recognition

Apply logic

Set parameters

Sort targets

Distribute or transfer assets

Coordinate aircraft

Report casualties

Request replacements

Control inventories

Source: Airland Battle 2000, pp. D-1 and D-11.

Figure II-5

- Military implications in space
- The human dimension
- Joint air and ground operations
- Worldwide command and control of forces
- Leadership
- Low intensity conflict and terrorism
- Organization of air and ground forces for combat
- Role of air support (18).

Space (battlefield surveillance), the human dimension (man-machine interface, computer-assisted instruction, simulations), air and ground operations (sensors, smart weapons), worldwide command and control of forces (communications), and organization of air and ground forces for combat (C^2, communications, tactical fusion) will all require software.

b. Air Force 2000

Air Force 2000 identifies ten highest priority military systems capabilities for the year 2000. They are:

- survivable enduring strategic forces
- remote surgical strike of fixed targets
- space offensive and defensive capability
- aerospace warfare management
- operations in a chemical/biological/nuclear environment
- adequate sortie generation
- penetration of enemy airspace
- operations independent of night and weather conditions
- mobile target kill
- rapid global force projection (19)

For each capability, Air Force 2000 discusses mission and needs, concepts and improvements, and the required technologies to achieve the capability. Figure II-6 summarizes the technologies required for each capability. Of the fifteen technologies listed, ten require software.

Information processing, as described by the Air Force, includes the science of artificial intelligence, other forms of computer processing, and software production which is so vital to the use of computers. A rudimentary application of artificial intelligence would be to "see" a group of objects, match the pattern with a limited set of master patterns in priority order, and select the best object for attention. A more sophisticated and difficult application of artificial intelligence is the gaming, option selection, and decision support necessary for C^3I (Command, Control, Communications, and Intelligence) at the national level. Artificial intelligence will make smart weapons truly smart enabling them to select the most important good decisions based on massive amounts of information in extremely short periods of time. These decision functions require an enormous amount of software. The efficient development of such software is essential to achieving the potential offered by the use of computers especially as software becomes more complex, expensive and difficult to check and maintain (20).

The challenge of sensor technology is to achieve reliable operation at night and in all weather, to automate such functions as sensor control and target recognition, and to integrate sensors with each other and the remainder of the avionics software suite (21).

In the area of supportable electronics, redundancy and self-checking need to be incorporated so that failures can be found and circumvented both in maintenance and during system operation. By emphasizing standardization, multiple use of hardware and software designs can be made resulting in simplified modification and upgrading. In unmanned systems, the "watchdog" function can be automated, which along with greatly improved component reliability will yield high "on-the-air" rates over extended periods (22).

Signal processing technologies to transmit or receive signals vital to communications, sensors, IFF (identification of friend and foe), and threat warning systems also require software. Signal processing is a digital technology where signals are sampled, converted to binary values and fed into a computer. Sophisticated coding schemes, cancellation of interference, sorting of multiple signals and other processing functions can be implemented in software allowing for modification by reprogramming without hardware changes (23).

Dormant guidance systems in ICBMs should be capable of rapidly responding with minimal platform settling time and minimal degradation in accuracy. Rendezvous of anti-satellite

AIR FORCE 2000 TECHNOLOGY MATRIX

TECHNOLOGIES / MILITARY SYSTEM CAPABILITIES	INFORMATION PROCESSING	SENSORS	SUPPORTABLE ELECTRONICS	STEALTH	LASER TECHNOLOGY	PROPULSION	SIGNAL PROCESSING	DORMANCY	RADIATION HARDENING	STOL	MUNITIONS	ADVANCED COMPOSITES	SPACE STRUCTURES	MANEUVERING REENTRY VEHICLES	SPACE POWER
SURVIVABLE ENDURING STRATEGIC FORCES	●		●	●	●	●	●	*	●	●		●			
REMOTE SURGICAL STRIKE OF FIXED TARGETS	●	*	●	●		*	●		●		*	●		●	
SPACE OFFENSE AND DEFENSE CAPABILITY	●	*	●	●	*	●	*	●	●		●	●	*		●
AEROSPACE WARFARE MANAGEMENT	*	*	*			●	*	●	●			●	*		●
OPERATIONS IN A CHEMICAL/BIOLOGICAL/NUCLEAR ENVIRONMENT		*	●						●		●				
ENSURE ADEQUATE SORTIE GENERATION RATE			●			●				*		●			
PENETRATE ENEMY AIRSPACE	●	*	*	*		●	●		●		●	●		●	
OPERATIONS INDEPENDENT OF NIGHT AND WEATHER	●	*	*	●			●		●		●				
MOBILE TARGET KILL	*	*	●	●	●	●	●		●		●			●	
RAPID GLOBAL FORCE PROJECTION	●		*			●				●		●			

* - Important   o - Applicable

Source: Air Force 2000 Executive Summary, p. 17

Figure II-6

weapons and escort satellites may require dormant or semi-dormant electronic, guidance, power and propulsion systems capable of highly reliable quick response (24).

Short take off and landing (STOL) technology requires advances in aerodynamics, propulsion, structures, control systems, and landing gear. Intelligent (i.e., software controlled) fire, flight, and engine controls are integral to STOL technology (25).

Of the fifteen technologies enumerated in Air Force 2000, it can be inferred that ten require software. Eight of the ten military system capabilities listed will use information processing and thus, software, directly. All of this software will be characterized by increased complexity (systems must be more accurate, autonomous, able to perform rapidly and continuously in all conditions, highly integrated and use artificial intelligence techniques), volume (virtually all weapons systems and aircraft will require software), and criticality (reliability of this software is crucial to mission success especially during autonomous operations).

As Air Force 2000 points out "traditional measures of reliability such as mean time between failure (MTBF) do not adequately characterize the reliability of systems whose failures are undetected until the systems are used in earnest (26)."

c. __Focus 21__

The "Technological Opportunities for Focus 21" Appendix to the __Focus 21 Working Paper__, a joint Army Air Force effort, specifies broad technological requirements for the 21st century (27). It describes the current capabilities and limitations and technology potential to improve capabilities for ten functional areas: Surveillance, Target Acquisition and Damage Assessment; Warfare Management (C^3I); Operation in CBN (Chemical/Biological/Nuclear) Environment; Operation Independent of Night and Weather Conditions; Force Deployment/Mobility; Electronic Warfare and Countermeasures; Target Engagement and Kill; Improved Force Efficiency; and Survivable, Enduring Combat Forces. Fifteen critical technologies for the next 20 years are identified:

Processing Technology (Information, Displays, Signals)
Directed Energy
Airframe Aerodynamics (Aircraft and Projectile, Control Force Generation)
Propulsion
Fuse and Warhead Technology
Mobile Electrical Power
Simulation, Training and Human Factors Technology (Man-Machine Interface)
Biotechnology
Electronic Systems Supportability
Electronic Devices
Signature Reduction Technology
Cooperative and Non-Cooperative IFF
Sensors
Communications and Data Transmission(28)

Of these, ten or two-thirds have future software requirements. For example, electronic system supportability requires the development of fail soft, fault-tolerant, reliable electronic subsystems, high throughput circuitry and the application of artificial intelligence techniques. Software for reliable, fault tolerant, fail soft avionics subsystems must allow missions to continue when portions of the base maintenance and support facilities have been destroyed. A major technological challenge will be the development of techniques to ensure reliability of components not regularly tested (29).

In the area of cooperative and non-cooperative IFF (identification of friend and foe), software to provide weapon systems with a near perfect identification and target classification capability and optimized for autonomous operations needs to be developed (30).

Communications require the development of standard computer language, formats, protocols, and hardware to permit more efficient use of the communications resources and provide greater flexibility (31). Lightweight smart antennas with software to automatically null out undesirable signals and use the total spectrum from optical to very low frequencies also require development.

In _Focus 21_, the Air Force identifies the following five highest payoff technologies of the technologies listed in _Air Force 2000_ and required for each capability:

- o Information Processing Technology
- o Sensor Technology
- o Supportable Electronics
- o Stealth
- o Laser Technology (32)

All of these require future software development. While the first two are software intensive, the last three, supportable electronics, stealth, and laser technology will also involve software control.

d. <u>AFSC Vanguard Planning Summary</u>

Vanguard is the Air Force Systems Command (AFSC) structure for planning future research, development, and acquisition activities. The Vanguard relates Air Force objectives to technology opportunities and options for future weapons systems spanning the time period from the present until 20 years in the future. The AFSC Vanguard Planning Summary is published annually.

Vanguard includes ten master mission areas: strategic offense, strategic defense, tactical air warfare, command, control, and communications, tactical reconnaissance/intelligence, war readiness material, space, mobility, electronic combat, and defense-wide and technology base activities. Each mission area is described in terms of mission objective, tasks, scenarios/threat, and current/baseline capability. Technology, development, and production goals for the mission area follow (33).

The technology and development goals for each mission area were counted and analyzed for software development components. Figure II-7 shows the ten mission areas, the number of technology and development goals identified for each, and the total number and percentage that require software. On average, 71% of these future goals will require software. When looking at the last mission area, Defense-wide and Technology Base Activities which is broken into near and far term technologies, the percentage of technologies requiring software is greater for far term technologies than near term technologies, implying that more future technologies will require software than is required at present.

Four of the five far term technologies listed will require software or be under software control. They are: Information Processing Technology, Microelectronics Technology, Directed Energy, and Sensor Technology.

Some of the data used to compile Figure II-7 are classified and therefore cannot be included here.

AFSC Vanguard Planning Summary Technology and Development Goals

	Total Goals	Goals Req. Software	%/Goals Req. Software
Strategic Offense	9	9	100%
Strategic Defense	25	19	76%
Tactical Air Warfare	19	12	63%
C^3	11	8	73%
Tactical Reconnaissance/ Intelligence	8	7	88%
War readiness material	19	8	42%
Space	6	4	66%
Mobility	11	6	55%
Electronic combat	14	12	86%
Defense-Wide and Technology Base Near Term Technologies	16	8	50%
Far Term Technologies	5	4	80%

Source: AFSC Vanguard Planning Summary

Figure II-7

e. <u>Navy Command and Control Plan</u>

The <u>Navy Command and Control Plan</u> presents an overview of global Navy strategic and tactical commitments, derives C^2 (command and control) capabilities in relation to warfare areas and support tasks, and assesses C^2 capabilities. The plan recommends actions to be undertaken over the next ten years to meet these requirements (34). Since C^2 architecture is heavily reliant on data processing and software, many future software requirements are implicit in future C^2 requirements.

The plan identifies 122 actions. Of the programmatic actions, 34 are new, 9 are modifications, and 16 are to existing programs. The remaining 63 actions are procedural (develop plans, define requirements, establish data links, etc.). Plans of action cover the following areas: strategic connectivity, submarine communication restrictions, communications vulnerability, sensor system coordination, command facilities, interactive data bases, multisource correlation, navigation integration, non-NTDS (Naval Tactical Data System) platform integration, and electronic warfare.

Of the 59 programmatic actions, 43 (or 73%) involve software development, modification, or integration. Appendix F lists the individual programmatic actions requiring software. Figure II-8 shows the numbers and percentages of new, modified, and existing program actions that require software. More new and essential existing programs require software than do modifications to programs. As to be expected in C^2, most of the software development is for communications, navigation, sensors, and command and control systems.

Navy C^2 Programmatic Actions Breakdown by Software Requirement

	Total	Actions Req. Software	% Req. Software
New Programs	34	28	82%
Modifications to Programs	9	2	22%
Essential Existing Programs	16	13	81%

Figure II-8

f. **DTST Ballistic Missile Defense (BMD)**

The Defense Technologies Study Team (DTST) Volume V report forecasts software requirements in support of the proposed ballistic missile defense (BMD) emphasizing battle management, communications, and data processing (35). The DTST report (Volume V), discusses the problems relating to battle management and survivability of key functions in an adverse environment. Many of the critical functions related to battle management, such as command and control, ordnance control, and weapon systems are highly dependent on software. Consequently, the software component remains a central aspect in meeting future requirements as outlined in the report.

The DTST Volume V specifically emphasizes that software requirements for the ballistic missile defense will exceed any other single system currently in operation. The report indicates that the proposed ballistic missile defense (BMD) software component will exceed current systems such as Safeguard by three to five times and will be larger and more complex than those previously developed. Also, as proposed, the BMD software component, to be effective, must meet more stringent controls.

The DTST report also indicates that there is an added layer of complexity in predicting BMD software requirements because as proposed the system will include a large number of geographically distributed computer resources, including those that are satellite based. In addition these space based resources must be capable of surviving a hostile environment. More importantly the software must be able to cope with temporary failures in parts of the system, that is, the software must be able to operate even if portions of the system are destroyed.

The report argues that the BMD battle management system demands special managerial and technical controls as well as development of specific software tools. In addition the report concludes that automated tools are needed to help interject controls and management oversight into the development of the complex BMD battle management system and its software.

In order to identify requirements for such a system the report suggests that automated tools be developed to permit "a formal set of requirements to be specified and verified for completeness and consistency." Consequently the automated development is predicated on development of appropriate management controls, verifing that implementation conforms to specification, and capturing the "corporate memory, that is maintaining a sufficient grip on salient events and activities."

The report also comments on maintaining and upgrading the software for such a complex system. It predicts that tools will be needed to support the development process and ultimately will be useful to support the maintenance effort associated with software.

The report reaches several major conclusions. The battle management system will, through its software, define and control the functioning of the entire defense and, thus, define its effectiveness and establish performance requirements for weapons and sensors. Three of the key conclusions follow.

<u>Conclusion 1</u>

"Specifying, generating, testing, and maintaining the software for a battle management system will be a task that far exceeds in complexity and difficulty any that has yet been accomplished in the production of civil or military software systems."

<u>Conclusion 2</u>

"The battle management system and its software must be designed as an integral part of the BMD system as a whole, not as an applique."

Conclusion 3

"The problem of realistically testing an entire system, end-to-end, has no complete technical solution. The credibility of a deployed system must be established by credible testing of subsystems and partial functions and by continuous monitoring of its operation and health during peace time."

There are two battle management functions that the Panel judged to be absolutely critical to the safety, credibility and effectiveness of a BMD system:

- o Authorized release of weapons, and
- o Ordnance safety during peacetime and testing.

These two functions are even more critical because a significant portion of a BMD system will be highly automated and will be operating unattended in space.

C. SUMMARY

Of the DoD systems mentioned in the FY 85 RDT&E budget that begin in 1985 or later, at least 80% require software. DoD long range planning documents call for future technologies and capabilities, at least 70% of which will require software.

DoD systems currently under development and described in future long range plans require software that is both more massive and more complex than software developed previously. The LHX automated cockpit is an example of software development that has never before been done. Software in future battlefield management systems will perform more functions, and handle larger quantities of more types of data, more rapidly that systems of today.

The increasing functionality of software demands a high degree of software reliability. Software on the X-29 forward swept wing aircraft, the A6-E Intruder aircraft, and remote submersibles is critical to system operation. This is because a human is not present to intervene, as in the case of remote submersibles, or incapable of reacting fast enough as in the case of the X-29 where flight control software makes some 40 adjustments each second in order to stay airborne. Software on the A6-E permits the pilot to navigate and drop bombs in bad weather, functions he would otherwise be incapable of performing. Systems of the future must be accurate, autonomous, able to perform rapidly and continuously under all conditions. All of these characteristics mandate reliable and adaptable software.

Future plans project that systems such as Air Defense systems will become increasingly modular in design to allow for product improvements without major system redesign. Since software is becoming increasingly responsible for system flexibility and functionality, the ability to rapidly and reliably modify software is significant.

Thus, the significance of the software component to the achievement of the proposed DoD plans. But with the exception of the DTST Ballistic Missile Defense Volume V, the plans reviewed made little explicit mention of software technology improvement. Creating an appropriate software environment, the report notes, is the key to meeting the systems goals. To support these developments several proposed and on-going DoD efforts are directed at improving DoD software development. Specifically the report indentifies the (STARS) Software Technology for Adaptable, Reliable Systems Program, the Ballistic Missile Defense Advanced Technology Center (BMDATC) activities, and the Defense Advanced Research Project (DARPA) Artificial Intelligence-based efforts (36).

REFERENCES

(1) DoD Digital Data Processing Study - A Ten Year Forecast, Electronic Industries Association, Government Division, Washington, D.C., October 1980, UNCLASSIFIED.

(2) The Military Software Market in the U.S., Report Number 690, Frost & Sullivan, Inc., New York, June 1979, UNCLASSIFIED.

(3) MIL-STD-1679A, Weapons System Software Development, 22 October 1983

(4) N. Murray, A. Hopkins, J. Wensley, "Highly Reliable Multiprocessors," AGARDograph No. 224, Integrity in Electronic Flight Control Systems, edited by P.R. Kurzhals. Advisory Group for Aerospace Research and Development, April 1977, pp. 17.1-17.6.

(5) Airland Battle 2000, Headquarters U.S. Army Training and Doctrine Command, August 1982, p.1, UNCLASSIFIED.

(6) Ibid., p. H-1.

(7) Ibid., p. B-3.

(8) Ibid., p. G-5.

(9) Ibid., p. G-11.

(10) Ibid., p. D-6.

(11) Ibid., p. D-11.

(12) Ibid., p. D-14.

(13) Ibid., p. D-15.

(14) Ibid., p. F-13.

(15) 1984 Weapon Systems-U.S. Army, Deputy Chief of Staff for Research, Development, and Acquisition, 1984, p. 129, UNCLASSIFIED.

(16) LHX Program Office, Army Aviation Systems Command, St. Louis, MO, Mr. Bob Tomaine, AV 693-1268.

(17) Op Cit., 1984 Weapon Systems-U.S. Army, p. 129.

(18) Major Mike Kendall, "Briefing on Airland Battle Doctrine and Army Futures Concept," Deputy Chief of Staff for Operations and Plans, Force Development Directorate, Doctrine Force Design and Systems Integration Division, 15 May 1984, UNCLASSIFIED.

(19) Air Force 2000; Air Power Entering the 21st Century, U.S. Air Force Office of the Chief of Staff, June 1982, p. 201, SECRET.

(20) Ibid., p. 397.

(21) Ibid., p. 398.

(22) Ibid., p. 398.

(23) Ibid., p. 401.

(24) Ibid., p. 401.

(25) Ibid., p. 405.

(26) Ibid., p. 210.

(27) Focus 21 Appendix-Technological Opportunities for Focus 21, Department of the Army and Department of the Air Force, Working Paper, p. 50, SECRET.

(28) Ibid., p. 31.

(29) Ibid., p. 36.

(30) Ibid., p. 37.

(31) Ibid., p. 39.

(32) Ibid., p. E-2.

(33) AFSC Vanguard Planning Summary, DCS/Plans and Programs, HQ Air Force Systems Command, December 1983, SECRET.

(34) Navy Command and Control C^2 Plan, Office of the Chief of Naval Operations, Director Command and Control, March 1983, p. ES-1, SECRET.

(35) Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles--Volume V: Battle Management, Communications, and Data Processing. Brockway McMillan, Panel Chairman, Defensive Technologies Study Team, Alexandria, Virgina, October 1983, UNCLASSIFIED.

(36) Ibid.

CHAPTER III CURRENT STATE-OF-PRACTICE

A. INTRODUCTION

In 1982, a Joint Service Task Force on Software Problems surveyed the entire history of prior studies and categorized the difficulties that DoD faces in exploiting the full advantage of computer technology. The Task Force drew the following conclusions:

- Software represents an important opportunity for the U.S. military mission.
- Technological leadership in software use and development is a major factor in maintaining military superiority.
- The current state-of-practice in DoD software development and support has potential adverse effect on the military mission.
- No "single problem" exists that can be overcome with a single solution.

This chapter describes the current state-of-the-practice in DoD software development and support. First, the nature of software development is described. This is followed by a discussion of the results of a data gathering effort that was conducted to determine the software engineering practices currently applied to the development of defense systems software. Detailed data presented includes those related to project management, pre-software development activities, software requirements definition, design, coding, and integration. In addition, testing and evaluation, configuration control procedures, and capabilities of software support environments are discussed. The summary presents major conclusions that can be drawn from the data.

B. NATURE OF SOFTWARE DEVELOPMENT AND MAINTENANCE

Characteristics of DoD mission-critical systems include:

- o large-scale, real-time, and fail-safe operation,
- o long life with continual changes,
- o development by large team and maintenance by a different organization,
- o co-existence with older systems and interfacing with unique hardware

Such system characteristics tend to provide challenges as well as constraints for the software development process.

The acquisition of major DoD systems is governed by the 5000 series of DoD directives and instructions. This policy is further amplified and implemented in regulations that are particular to each of the Services. The ultimate responsibility for adherence to the policy, however, resides in the project offices. Military standards have been developed for application on development contracts to help ensure that minimum requirements are met.

Military standards that define the software development process include the proposed DoD-STD-SDS on Defense System Software Development. It describes a computer software development cycle that comprises a set of activities and their associated products and reviews. The activities include:

- o Software requirements analysis: the establishment of a complete set of functional, performance, and interface requirements.
- o Preliminary design: the development of a modular, top-level design from the software requirements.
- o Detailed design: the development of a modular, detailed design.

- o <u>Coding and unit testing</u>: the coding and testing of each unit comprising the detailed design.
- o <u>Software integration and testing</u>: the integration of units of code, informal testing of aggregates of integrated units, and formal testing, as required.
- o <u>Software performance testing</u>: the conduct of formal tests including the recording and analysis of results.

The presentation of the data gathering results can be traced to these activities with a few minor exceptions. Data related to the preliminary and detailed design activities are combined, as are data related to test and evaluation activities. In addition, general information about the projects examined and data describing activities that occur prior to the initiation of the software development and throughout the life of the projects are included.

C. PROJECT CHARACTERIZATION

This section provides brief overviews of the projects examined, the applications implemented in software, the magnitude of the software development efforts, and the software development life cycles employed. Since anonymity was promised to all subjects who cooperated during the data gathering effort, projects and organizations will not be identified in the following discussion.

1. <u>Project Overviews</u>

Detailed data were gathered on eight systems. Six of the systems were major defense systems (3-Navy, 1-Air Force, 1-Army, 1-Navy/Air Force). The remaining two systems (1-NASA, 1-commercial) were examined to determine if any significant differences exist between the practices applied to defense system developments and those applied to other efforts.

The individual development efforts for the projects examined were initiated in the late 1960's (2), the early 1970's (3), and the late 1970's (3). Five of the systems are fully operational at this time, two are operational but do not meet the original expectations for the systems, and one is currently undergoing follow-on testing to determine suitability for deployment. The shortest development effort spanned a period of two years; the longest required more than fifteen years.

2. Software Applications

Figure III-1 categorizes the functions implemented in software for each of the systems.

Although it is difficult to assign quantitative values to the percentage of system functionality embedded in the software, in all cases it was agreed that without the software the systems could not meet essential mission objectives. Furthermore, in all cases, the software was responsible for the implementation of functions that had never been attempted before -- even in hardware.

The partitioning of systems functions between hardware and software was usually based upon past experience, whether gained during the concept development phase or on similar system developments. In one case, the hardware was designed first and thereafter influenced the functionality of the software. In another case, the software was designed first (prior to the choice of hardware). However, because of the lack of understanding of the requirements and the migration of functionality from the hardware to the software, the final size of the software was double that originally planned.

3. Magnitude of Software Development Efforts

Where available, estimates of the cost of the software ranged from a low of $20 million to a high of $180 million.

FUNCTIONS IMPLEMENTED

Number of Systems	3	1	1	1	1	1
Tracking	x		x			
Guidance & Control	x	x	x	x	x	
Navigation	x	x		x	x	
Digital Filtering/ Image Processing	x	x	x			
Computation	x	x	x	x	x	x
Command & Control/ Information Management	x	x	x			
Built-in-Test	x	x	x		x	x
Applications Support	x	x				
General Data Processing						x

Figure III-1

These are only estimates since software is not usually costed on a line item basis. Another comparison of system/software development effort can be made based on the number of people involved. These values ranged from 20 persons for one system to a peak of 275 for another system. On four of the projects, the software developers had experience with similar systems in the past. In one case, it was the developer's largest by at least two orders of magnitude. Only two of the development organizations had groups specifically charged with looking for and inserting new technology to improve the practice of software engineering in the organization. In one case, different managers would investigate the possibilities "from time to time based on need."

Software size is another area where standard measures are difficult to find. Some organizations measure software based on the number of lines of code. Of these, some include comments; others exclude comments. Yet other organizations base their measures on the size of the object code. When comparisons are made, care must be taken that the word sizes used to relate the object code size are consistent (i.e., 16 bit, 32 bit). Also important is whether or not the measures reflect the size of the operational or mission software alone or whether support software is included. Sizes of software reported for the systems under examination ranged from 11,000 to 900,000 lines of code including comments for mission software only. After adding support software, the original 900,000 lines of code increased to 2,150,000 lines. When measuring the size of the object code, values reported ranged from 16K words of operational software for one system to 7 million words including support software for another system. The percentages of code produced during a system development that were actually delivered to the customer ranged from 25% to 100%.

Finally, the percentage of code in a system that is reused from other development efforts was addressed. Four of the systems examined reported no reuse of code. One reported that approximately 40% of the operational software (80% when including support software) was reused from previous development efforts. Another system reported reusing 18 out of 25 modules from a previous system.

4. Software Development Life Cycles

For six of the systems, initial plans were to follow the standard waterfall development process (requirements definition - design - code - test). Of those six, three later changed over to some type of incremental development or iterative enhancement. Another of the six followed a parallel development where code was written from the requirements documents and the design was developed "after the fact." Only one of the six followed the original waterfall plan to fruition.
It should be noted however, that that system is now in a support phase where it is enhanced on a continuing basis to meet the changing threat and incorporate new technology. The other two systems planned up-front to follow a "build a little, test a little" approach to software development -- one beginning with the detailed design phase of development; the other during the coding phase.

D. PROJECT MANAGEMENT

In this section, techniques used to estimate software development costs, staffing requirements, and milestones and schedules are presented. Data collection and tracking mechanisms and procedures for updating estimates are also discussed.

1. Costs

Cost estimation techniques applied on these programs were based on historical data, expert judgement, and analogy. Some of the combinations employed were:

- The size of the software and the productivity rate of the organization were estimated based on experience gained during the concept exploration phase, expert judgement, and analogy to other development efforts. In addition, both bottom-up and top-down techniques were employed independently to verify the estimates. This was the first time these techniques were used. Prior estimates had always been those developed on the "back of the envelope".
- Commonly available costing models were applied (Boehm's COCOMO, Putnam's SCM), and results tempered with engineering judgement. This particular organization has been using this method for the past 3 years. Previously, estimates had been totally based on historical data.
- In-house "rules of thumb" were applied. The cost of key functions was estimated and combined bottom-up. Complexity of the units was estimated and factored into the cost. Lifecycle factors were also included. These estimation efforts benefited from the use of internal tools that were constructed and tailored to the project based on the experiences of the contractor.
- Cost of software was estimated based on a previous similar project. Specific differences between the past and current project were taken into account in the estimation process. In addition, a small-scale model software development was undertaken to further estimate the effort.

Regardless of the technique employed, results consistently over-estimated productivity and under-estimated required effort.

Projects also reported that data was collected on a continuing basis during development efforts so that estimates could be updated and refined. Update cycles ranged from estimating the cost to completion on a monthly basis to updating the cost of the overall project once a year as required by the budgeting process.

The amount of data collected for tracking purposes varied greatly by the organizations. In one case, "programmers' daily activities" were tracked. The following describes the other extreme:

- Software budget measurables (could be directly linked to a deliverable) and the level-of-effort budget spent were tracked and used to calculate the variance between the actual costs and the estimated costs. In order to determine what items to monitor, each major program in the Work Breakdown Structures (WBS) was assigned a cost account for each of its deliverables (specification documents, design documents, code, etc.). These accounts were further subdivided into work packages and measurable work package tasks (designing, coding, testing, etc.). Hours and dollars were budgeted and tracked on a monthly basis at the level of the measurable work package tasks. Level-of-effort tasking was kept to a minimum to facilitate accurate cost accounting.

At least two of the organizations had developed tools that were tailored to their specific business environments for the purpose of tracking and reporting desired cost data. One organization had built a database of cost information during previous projects. This was used in the cost estimation process. Another organization was building a database for future use that contained records of the level and type of effort for each software component handled on a weekly basis. Yet another group was using the actual data to "fine tune" the internal cost estimation models and the parameters used. This group felt that the costing process was "somewhat guesswork" -- the key was to collect actual data and refine the estimates as the project progressed.

Half of the organizations experienced severe cost overruns on their projects. In one case, discrepancies between estimated and actual costs were attributed to programmatic problems, changing requirements, and management inefficiencies. In cases where the software was delivered within cost, the importance of gathering data and updating estimates throughout the program development was stressed.

2. Staffing

Approaches used to estimate staffing needs were based upon the cost estimates derived previously and the perceived project needs or demands. In one organization, individuals were reassigned and temporary employees hired on an "as needed" basis. This method resulted in an increase, in a very short period of time, from approximately 20 people to a peak of 70 people working on the software design. It was later demonstrated that the "need" was for time -- not people.

Similarly, another organization began staffing its project to meet the schedules. However, the practice was discontinued because of the instability injected into the development process, especially when requirements changed. Thereafter, a "flat" staffing profile was maintained (minor modifications were made on a quarterly basis; major modifications once a year). The use of a constant workforce substantially influenced the determination of project schedules and priorities.

A third less extreme approach, involves the use of matrix management. In this organization, a somewhat stable pool of personnel exists from which to draw. Although this may still introduce some instability into the development process, the results are not as severe as hiring new or temporary personnel into a strange environment.

None of the organizations examined reported any special tools or data collection procedures related to project staffing. Any tracking which occurred was a by-product of monitoring cost and schedule.

3. Milestones and Schedules

The level of granularity in the schedules and the approaches employed to develop them varied among the organizations. The following paragraphs describe some of the techniques used.

- On the first program, the government's major milestones (PDR's, CDR's, etc.) were the software development milestones. Experiences gained reflected the need for day to day tracking of all program elements and close monitoring of schedules to allow early recognition of problems.

- The second program determined schedules and milestones based upon the available money. Procedures have recently been updated such that demonstrated progress also influences the program schedule. This program has had significant problems throughout its development. Although in-house tools now exist to aid in the scheduling process, their application has suffered from a lack of confidence by program management personnel.

- The third program made a very strong commitment to a final delivery date. Intermediate milestones were derived in an ad hoc manner as a result of task planning, manpower analyses, and prototyping performed to estimate the size and complexity of the effort. As development progressed, numerous changes in requirements and development strategies voided the original schedule. In addition, the prototyping did not accurately reflect the complexity of the production system. The original delivery date was met, but with a poor quality product with reduced capabilities. This situation was remedied by delivering periodic enhancements to the product until requirements were satisfied.

o The fourth program's schedules and milestones were based on program demands and model (PERT/CPM) projections. "Conceptual schedules" were updated once a year. Development schedules were updated quarterly. Detailed schedules were updated as required. Occasionally, overtime or the cancellation of planned activities was necessary to meet the schedules, however, the overall software development was regarded as an example that software is not always late, short of capabilities, and short of quality.

o The final program constructed a matrix where the rows were the deliverables and the columns were the phases of the software development lifecycle. This was placed on top of a calendar which showed the hardware availability dates, major government milestones, and the final software delivery date. After allowing six months for integration prior to the final demonstration, the time available for the software development could be derived. Several evenly spaced, incremental deliveries to the government were scheduled such that the most difficult and critical capabilities would be developed first. Those areas which had unstable requirements were scheduled later in the development whenever possible. For each incremental delivery to the government, several intermediate deliveries from the software group to the systems engineering group within the organization were also scheduled. In addition, detailed schedules were developed such that each person had an average of one milestone per week.

The "micro-schedules" changed weekly, but all incremental deliveries to the government occurred on time (in some cases, requirements satisfied by an increment were cut back to allow on-time delivery). The final delivery of the total software system also occurred on schedule. Items monitored to track progress included planned and actual completion dates for coding, unit testing, string testing, etc. of each function. In addition, the "earned value" of the product was monitored. When determining the earned value, an element was not included in the calculation unless it was "complete" (i.e., an element was either 100% coded or 0% coded; 90% was not valid). Tools used to develop schedules and track the progress against them were essentially the same as those used to monitor costs and staffing. PERT had been considered for use and rejected because of the amount of time required to do updates.

E. PRE-SOFTWARE DEVELOPMENT

Prior to software development, numerous system level activities are performed. In terms of the DoD major system acquisition process, the Concept Exploration Phase (Phase I) and the Demonstration and Validation Phase (Phase II), which culminates in the definition of the system requirements, may both be complete prior to the initiation of the software development process. This section discusses concept exploration and system requirements definition activities.

1. Concept Exploration

The basic technique used for concept exploration was prototyping. Variations employed include prototyping the system, prototyping major subsystems, and competitive prototyping. In addition, trade-off studies, feasibility studies, and architecture studies were described as activities undertaken during the concept exploration phase. Simulations and simulators were the primary tools used to support these analyses.

One project noted the importance of planning for the iterative nature of systems and requirements definition. In this case, the formal completion of concept definition was proclaimed per the schedule. In actuality, however, redefintion and iteration of the requirements continued throughout the life of the project. Another organization commented on the difficulty of discriminating between the "real" requirements for a system and those that are "passing fancies".

2. System Requirements Definition

System requirements definition tends to be a continuation, or a more formal instantiation, of the concept definition phase of development. The formality appears in the form of documentation produced for the purpose of contractually specifying a system which is to be built. It is also at this time that the requirements begin to be reviewed by the appropriate interested parties.

Six of the projects followed the process described above. In one case, this phase consisted of simply stating constraints on the accuracy of primary system functions. Another project specified the software requirements first, utilizing a top-down approach to developing a partitioned hierarchy. The top level of the hierarchy consisted of the architecture, followed by the functions, the functional relationships, and, finally, the detailed descriptions. This method was similar to that employed by the organization on previous efforts, but more formal. It spanned several years and involved several hundred people.

The primary notation used to describe the system requirements was English text (no formal requirements languages were used). One project used system data flow charts. Another utilized narrative text supported by optional flow charts, interface tables, and dependency diagrams. Tools used in support of this process included word processors. The deliverables associated with the system requirements phase were the Type A-System Specification and the Type B1-Prime Item Development Specification.

F. SOFTWARE REQUIREMENTS DEFINITION

Four of the projects described the software requirements definition phase as an outgrowth of the system requirements definition phase, which itself had been a continuation of the concept definition phase. In one case where there was a change in activities between the concept definition phase and the system requirements phase, the software requirements definition activities were merged with those of the system. This was necessitated by the "software-first" approach that was being employed. For another system, the prime contractor received a draft software requirements document that was then updated by the prime's engineering group (rather than the software group) for changes in the requirements resulting from perceived changes in threat. Only one organization described a procedure involving detailed examination and study of the software-unique requirements. In this case, extensive modelling, simulation of algorithms, sensitivity analyses, and timing studies were conducted to refine the software requirements. In addition, formal procedures were put in place to facilitate the clarification of system requirements as necessary for the specification of software requirements. One additional project also used simulators at this point to verify the software requirements being developed.

The notation used to express the software requirements and the tools used to support this phase did not change significantly from previous phases. English text and text processing systems were the predominant technologies applied. In one case, however, it was reported that PSL/PSA was used. In another, NRL's SCR requirements specification methodology and notation is being employed for enhancements. Finally, a verification and validation contractor was applying a SREM-like tool to the software requirements produced by the prime contractor for its project.

During this phase, the review cycle normally associated with a software development begins. Two of the projects employed independent verification and validation (IV&V) organizations for the purpose of scrutinizing the software being produced. In one case, this function was performed by an independent contractor; in another, the future support organizations were involved. One project employed a verification and validation contractor (as noted above). In this case, independence did not exist between the V&V contractor and the prime since the V&V contractor was encouraged to deal directly with the prime contractor and even developed some of the software. When neither IV&V nor V&V organizations existed, quality assurance (QA) organizations supplemented the reviews performed by the development organizations. One organization commented, however, that the QA organization suffered from the "good old boys" syndrome -- to the detriment of the effectiveness of the reviews.

The minimum review, at this point, generally consisted of tracing each software requirement back into the system requirements document and constructing a traceability matrix. Other attributes which were searched for during the reviews included content, completeness, accuracy, and consistency.

The deliverables associated with this phase were the Software Requirements Specification, the Program Performance Specification, and the B5-Computer Program Development Specification (or Part I Specification).

G. DESIGN

Six of the projects examined used a top-down approach to design their software. Two of these were described as top-down modular design techniques; three as top-down structured design techniques. In one case, two levels of refinement were performed: from high level to detailed modules and from detailed modules to functions. Exception handling was also given a lot of attention. Another of the systems was designed using fault tolerant techniques: redundancy, checks on inputs, error detection and recovery routines, and reconfiguration and reload capabilities. This system also designed in a common system executive and common error handling routines. In order to facilitate future enhancements and the ability to operate in a degraded mode, each module composed either part of one or one complete element function. For one project, a large degree of importance was placed on avoiding timing and efficiency problems in the design of its asynchronous system. The approach employed was to attempt to design the system such that there were no time criticalities.

The two remaining systems did not supply great detail on their design processes. One simply stated that the software requirements for the functional areas were elaborated into functional specification documents. The other divided the requirements into functional modules and assigned them to different groups for implementation. The primary concern for this organization was the efficiency of the implementation.

The notation used to express the software designs included English text, program design languages (PDL) (4), flow charts (3), block diagrams (1), data flow diagrams (1), pseudo-code (1), hierarchical input-process-output diagrams (HIPO's) (1), and prototypes (1). Relatively few automated tools were available for use during the design phase. In general, developers had access to standard word processors and editors. One organization reported continuing its use of PSL/PSA. Another employed simulators to evaluate the prototype designs. The effort expended during this phase of the development averaged between 30 and 40% of the total software budgets.

The review cycle initiated in the software requirements definition phase continued in the design phase. Again, the predominant technique employed was to trace the design back into the software requirements document, and, in some cases, into the test planning documents to ensure complete coverage of the requirements during formal testing. One organization relied solely on the results of the formal government PDR's and CDR's for an evaluation of its design. Peer reviews were not held since the designers were felt to be highly qualified people.

Another held structured design reviews. One organization described formal design inspections which required 6 to 10 people for 2 to 4 hours when evaluating 100 to 1000 lines of design. At the conclusion of the inspection, a pass/fail decision would be made concerning the maturity of the design. Data was gathered during the sessions on the number and type of errors found and the effort required to conduct the inspection.

In two cases, in addition to the internal developers' reviews, QA organizations conducted reviews: one was non-technical, the other was concerned with the accuracy, consistency, and flow-down of the design from the requirements. Two organizations also had their designs evaluated by IV&V organizations. Supportability was the key concern for one of the projects since multiple organizations were planned to perform this function after deployment. The only metrics calculated which influenced the software design was that of mean-time-to-repair for one system.

The following comments were made about the formal government reviews:

Design reviews become instructional sessions when participants are not prepared.

User attendance at PDR's is beneficial -- it gives the users an opportunity to see what they will be receiving. User attendance at CDR's, however, can cause problems -- they may interfere with the progress being made by the analysts.

The deliverables associated with this phase of the development included the Program Design Specification, the Program Description Document, the Type C5- Computer Program Product Specification or Part II Specification , Data Base Design Documents, and Interface Control Documents.

H. CODING

Six of the projects examined were developed and coded using structured programming techniques. In addition, two described a top-down development, and one a top-down message flow development. In one case, it was noted that exceptions to the usual standards of structured programming were allowed when necessary because of "complex real time requirements." This organization also allowed multiple entrances and exits to the modules.

Coding standards applied on the projects addressed the following areas: liberally commenting the code, including preface blocks or preambles, labelling and naming conventions, maximum levels of indentation, and lines of code in a module. The most common source of these standards is MIL-STD-1679. In addition, one project required that each equation in the code be cross referenced to the requirements specification. Other standards applied pertained to communications protocols, and timing and core reserves. One organization reported that prior to 1981, no programming standards were applied on its project.

The progamming languages used included various dialects of CMS2, JOVIAL, FORTRAN, PASCAL, COBOL, SPL/SPL1, a special purpose high order language, and assembly language. Two of the projects reported using 100% assembly language for their applications due to timing and sizing constraints. In other cases, assembly language accounted for from less than 1% to 30% of the code. Two projects reported the requirement for a waiver whenever assembly language was used. In those cases, assembly language was only used for interrupts and I/O capabilities -- "whenever the compiler couldn't handle the job". For one project where support software was developed in addition to the mission software, "all languages available" were used in the development of the support software. The mission software was developed using a special purpose high order language (85%), supplemented by assembly language (15%). One group reported that this was their first use of a high order language; another reported that without the high order language, it would not have been possible to meet the projects' budget and schedule. On one project where only assembly language was used, the code originally contained no line numbers or comments. On the other 100% assembly language project, macros that enabled conformance to the principles of structured programming were used.

The automated support available on one of the projects consisted of an assembler. Two projects reported the use of editors, compilers, and debuggers. Three projects used preprocessors to enforce coding standards. One group felt that as the size of a project grew and the number of people increased, so did the need for standards and enforcement. Furthermore, it was felt to be impractical to attempt enforcement of coding standards through the use of audits -- automated tools were necessary. Other tools applied on these projects included listings formatters, global cross-reference and statistics generators, multitier libraries, "As-Built" documentation generators, and simulators. The documentation generator would produce narrative comments, structured flow charts, and module procedure flows based on the written code.

Six of the projects also described the use of code reviews to evaluate the implementation of the software. Three of these applied formal Fagan code inspections; two used peer walkthroughs; one used informal reviews by section leaders. In three cases, QA groups audited the code for compliance with standards. Two of these were formal audits using checklists;

one consisted of reviews of documentation and spot checks of listings. IV&V groups performed code analysis for two of the projects. In one case, this took the form of tracing the requirements into the code and vice versa. In the other, critical algorithms were independently derived mathematically and the corresponding code analysed. In one case, it was reported that the level of formality applied to the code review process varied with the size and complexity of the code.

The deliverables associated with this phase of the development included Program Packages (source and object code and listings, and cross-reference listings), and Version description documents. The amount of effort expended was reported in general to be 10-20% of the total software effort.

I. INTEGRATION

As described earlier, two of the projects planned to follow a "build a little, test a little" approach to software development. In both cases, the integration strategy was to release usable increments, each a superset of the previous release. One organization had a list of heuristic "do's and don't's" to guide them in the choice of capabilities to be implemented in an increment. Other integration strategies employed included top-down; bottom-up, one module at a time; and that of relying on experienced programmers to perform the integration task. Finally, one system described a process of migrating the software from a simulated environment to the operational enviroment. The four steps taken utilized:

(1) Actual software plus a lot of software diagnostics, minimal actual hardware.

(2) Actual software, actual hardware, a lot of diagnostics.

(3) Actual software, hardware, and users (in training).

(4) "Full-up" system.

It was felt that this approach to integration, though thorough, involved too much duplication of effort. In a seven year period, approximately 1,000 people were involved in the integration of this system.

J. TESTING AND EVALUATION

In general, the software developments included at least four stages of testing: unit or module testing, software integration testing, software/hardware integration testing, and acceptance testing. Additional levels of testing were conducted when the developments included incremental releases and when IV&V organizations were charged with conducting independent tests. In the following, we will describe the various levels of testing and strategies employed on the projects examined.

Seven of the projects described some form of unit or module testing. On two of the projects, the decision of whether or not to perform unit testing and, if performed, the strategy and extent of the unit testing was left to the prerogative of the individual programmers. The fear of releasing bad code for "all of the world to see" was felt to be enough incentive that an appropriate amount of testing would take place at this level. In another case, module testing was "combined" with software integration testing. After experiencing severe problems with this approach, the testing of individual modules was reinstated. This organization tried executing all paths at the subroutine level but decided that this was not cost effective. It noted that there was "room for improvement" in the area of module testing. Another project described its approach at this stage as testing to the detailed design documents. This organization initially skipped some module testing but eventually went back and performed the omitted tests. Finally, one group attempted to achieve 100% statement coverage during module testing, as well as performing stress and extreme value testing. However, no audit trails existed to allow the reconstruction of the tests or the verification that goals were achieved.

At this point, the two projects that had initially planned incremental developments inserted an extra stage of testing into their development processes. In one case, coverage analysis was performed. In the other, this level of testing was conducted by the programmers' managers prior to releasing the software from the department. This stage was followed by another additional stage of software integration testing. This second stage comprised the approval to proceed to the complete increment testing.

The next stage of testing is the common software integration testing or tests of "large" pieces of software. This level of testing is usually derived from the software requirements specification. For the systems undergoing incremental releases, this stage of testing was performed by the recipient of the increments (in one case, the prime contractor received the increments from the software subcontractor; in the other case, the government received the software from the prime contractor). For one of the non-incremental developments, a separate test group constructed test matrices and a compliance matrix. It also defined nominal and abnormal inputs for the testing at this stage.

Software integration testing is followed by software/hardware integration testing. This is usually formal testing based upon functional specifications and performed by contractors' engineering organizations or separate test groups. Two of the projects reported that QA personnel witnessed these tests; in one case, the customer also witnessed the tests. One group described this level of testing as testing every requirement until it is "OK".

The final stage of testing was system level testing. In two cases, this was described as extensive testing of system level functions by systems engineers. For one project, scenario tests were executed for all functions in simulation labs. For two other projects, boundary limit tests, stress tests, and error response tests were conducted in the process of evaluating the system functionality. In at least one case, this level of testing was a subset of previous tests, rerun to satisfy the final requirement prior to system delivery.

Independent testing performed by IV&V organizations was conducted for two of the projects. In these cases, the tests were designed to concentrate on areas of concern which were uncovered during documentation analysis, reviews, or analysis of testing performed by developers.

Four organizations documented the testing process with test plans, procedures, and reports for formal testing only (formal testing is usually the final system level testing and, in some cases, very high level integration testing). Two of the organizations also used test specifications. In one case, formal test plans were reviewed and approved by the customer; in another case, this was the responsibility of the QA organization. When formal testing was witnessed for one project, the observers recorded any anomolies and logged changes to the test plans and procedures. Only one organization described documentation of the module level testing process. This group maintained programmer's notebooks which contained unit test plans and procedures. However, it did comment that the level of formality in reporting testing information increased with the level of the test.

By far the most common tools used to support the testing process are simulators and stimulators. Unfortunately, these tools are usually project specific and cannot be applied elsewhere. Two organizations also described data extraction/reduction capabilities and file comparators applied during testing. One group used a data compare facility which included test scenarios and results expected for given time periods. Single organizations reported the use of unit test drivers, coverage analyzers, and homemade tools for test generation and execution.

Five organizations described error analyses conducted. Two groups categorized errors in terms of criticality (in one case, the categories described in MIL-STD-1679 were used). Errors were also tracked according to which test group found the errors, what level of testing was achieved, and the amount of time spent testing. One group also analyzed the trends in the number of errors discovered after new releases of the software. Two organizations specifically tracked and analyzed error reports for the purpose of isolating "problem areas" in the code or organization.

In addition to error analysis results, cyclomatic complexity metrics were used by two groups to guide the testing effort. Another set goals for mean-time-between-critical events and interrupts to determine when the software quality was sufficient.

K. CONFIGURATION CONTROL

The importance of configuration control in the software development process is generally recognized. Six of the projects described a similar, formal change control process. Each involved a form (computer program problem reports, software problem reports, program trouble reports, software trouble reports, software change proposals, or software change requests) and various authorization boards (Configuration Control Boards, Software Change Request Boards, Software Change Review Boards). The forms were used to provide information identifying the requested change or reported problem, its feasibility, its impact on cost/memory/timing, the recommended release in which the change should be implemented (if at all), and other information important to the decision and tracking process. The boards were responsible for reviewing the information, analyzing the changes and deciding when the change should take place. Three of the organizations had placed the configuration control forms online to facilitate reporting and tracking this information. In one case, any time the software was changed, a comment had to appear in the code referencing the configuration control form number which authorized that change.

Although the configuration control process described above was similar for the six systems, the items actually controlled and the point in the development process when the items were placed under configuration control differed. Examples follow.

o Only the source code was placed under formal configuration control. The software requirements specification was updated informally via the use of change forms. The design documents were not updated. This organization had originally planned to control all products but decided that this was too costly and impractical.

o Only the software requirements specifications were base-lined. This occurred when the organization was "ready". A "lesson learned" by this organization which was passed on during the information gathering discussion was that configuration management and control is needed for all aspects of a software development program.

o The software requirements specifications were placed under configuration control at the completion of the CDR. Source code was placed in libraries controlled by the QA organization after unit testing was complete.

o The software requirements specifications were placed under configuration control "early". Once a piece of software had completed unit testing and the initial stages of integration testing, it was placed in a test library which was under configuration control. After successfully undergoing the remainder of testing, the software is placed in a release library. The hierarchical library system used both controlled access to the pieces of software and tracked the changes made.

o A layered approach to configuration management consisting of six levels of control for interim and final products was employed. It was felt that, in this case, the iterative development necessitated a high level of technical involvement. Fifteen to twenty percent of the total project staffing was allocated to configuration management.

L. SOFTWARE SUPPORT ENVIRONMENTS

This section provides an overview of the hardware used during development and post deployment software support capabilities.

1. Hardware Support

Three of the projects used VAX 11/780's with VMS as host machines during their development efforts. One project used an IBM 370 with VM/CMS; another used an IBM 370 batch system with cards as the input medium. Another project recently acquired a VAX 11/780 with UNIX to replace its batch system. Two of the projects used the target machines, which were specifically designed to be embedded computers, as host machines during the development. The target machines included standard Navy hardware (AN/UYK-7's, AN/UYK-20's, and AN/AYK-14's), and other embedded computers built by IBM, Litton, and ROLM.

2. Post Deployment Software Support

Four of the projects examined are being supported after deployment by the original development contractors. One of the organizations is in the process of developing user friendly tools so that personnel who do not have an extensive knowledge of the system will be able to maintain it. Tools available to aid in the support phase include those used during development plus special purpose reconfiguration tools. These have been integrated to share common information and databases. In addition, maintenance specifications are being developed to capture corporate knowledge, communicate assumptions made during the design process, and supply guidelines for the maintenance process. Portions of the code will be redeveloped, rather than modified for future changes, only if there is a potential for major savings in the areas of memory usage or maintainability. This system has been operational for less than 5 years and is expected to remain operational for another 15-20 years.

The remaining four projects are being (or will be) supported by the government, in some cases with contractor support. On one of the projects, new tools were developed for post deployment support. It should be noted, however, that the tools developed do not support activities specific to the PDSS environment. In fact, it was felt that the same tools would have been beneficial during the initial development. Tools now available include an assembler, a memory loader/verifier, and simulation laboratories. In addition, the original documentation was of poor quality and has been upgraded or, in cases where it did not exist, created. This system has been operational for approximately 15 years and is expected to remain operational for another 5 years.

One of the other systems requires further discussion. This system is in the process of being transitioned from the development contractor's care to the government's care. Because of the applications implemented in the software, it will be supported by three government support activities. One of these activities has software development experience and the necessary equipment; another will be supporting software for the first time; and the third has experience, but little in the way of automated support. Though the support activities were named at the initiation of the project and performed IV&V activities on the software they were to support after deployment, it remains to be seen if this division of responsibility will be an effective way to approach the post deployment software support for this project.

M. SUMMARY

Of the eight systems examined, five of the development efforts are considered to be "successful"; three are not. Although strict cause/effect relationships cannot be derived from the data gathered, this summary highlights a variety of interesting aspects. For example, a common belief is that developments that utilize new technology will succeed when others fail -- the data does not support this conclusion. Furthermore, no major differences were discovered between the DoD developments and the NASA or commercial development. Each effort encountered difficulties that had to be dealt with. The solutions applied were not unique to the non-DoD world.

The capabilities implemented by software are becoming more varied and more critical to the successful accomplishment of the military mission. In each of the cases examined, the software was being used to implement functions that had never been attempted in the past. The susceptibility to requirements definition problems increases when a solution is being attempted for the first time. In spite of the importance of the requirements definition process and the great amount of influence that it has on the ultimate success of the program, few systematic techniques and little automated support existed for these activities. Although, as a minimum, the requirements are usually traced into the design, code, and sometimes tests, even this analysis is predominantly conducted with little or no automated support.

The single overwhelming commonality that existed among the systems investigated was the requirement to accommodate change. Refinements to problem solutions and changes in the environment in which the system will be deployed all require modifications to the software. As a result, whether planned or not, all software undergoes some type of incremental development or iterative enhancement.

Estimates of software costs, staffing requirements, and schedules do not accurately reflect the true needs: the sizes of the efforts are underestimated and the productivity of the people is overestimated. Errors in either of these estimates have similar results. Combined, the effects are even more serious. The estimates may in fact be accurate in a stable environment. However, as reported above, the environment of software development and deployment is not a stable one: the one true constant is that of change.

The technologies applied during software development can usually be traced to the requirements of the military standards that have been referenced in the appropriate contracts. In general, developers are not rewarded for exceeding the technological requirements of the standards, and these did not do so. The primary variance related to the use of technology in software development is found in the level of discipline and formality with which the technology is applied. Figure III-2 provides a summary of the technological state-of-the-practice encountered. Further aggravating the situation is the fact that, in many cases, software is not treated as a manageable entity. This may stem from a lack of understanding on the part of management and a general feeling of discomfort when dealing with software.

The transition of responsibility for software support from the developing organization to the Post Deployment Software Support activity does not usually occur without problems.

At times, the data gathering effort was hampered by the unavailability of data. There were two reasons for this: either the data was considered to be "sensitive" or it did not exist. Even when data was available, it was not always useful for comparison purposes, as was the case with the measurements of software size. The field of software development and engineering continues to be plagued by a lack of generally accepted quantitative measures.

The current state of practice is experiencing problems in meeting current requirements. Programs have difficulty defining requirements and requirements are constantly changing as software becomes increasingly responsible for implementing new functions. Other problems are in budgeting, staffing, and scheduling and with product quality.

CURRENT STATE-OF-PRACTICE OVERVIEW

Activity	Method	Formal	Automated
Project Management			
Cost Estimation	4	2	2
and Tracking	4	3	3
Manpower Estimation	4	1	0
and Tracking	3	2	3
Milestone Estimation	5	2	2
and Tracking	2	2	1
Systems Requirements	6	1	3
Software Requirements	4	2	2
High Level Design	8	7	2
Detailed Design	6	6	2
Code	5	6	8
Unit Test	7	1	7
Integration	6	4	8
System Test	8	8	8
Configuration Control	6	6	4

Method = Identifiable Method

Formal = Documented, Followed, and Verified Method.

Automated = Automated to some Degree.

Figure III-2

CHAPTER IV TECHNOLOGY CASE STUDIES

A. INTRODUCTION

We have reviewed the growth and propagation of a variety of software technologies in the hope that we can discover the natural characteristics of the process as well as the principles and techniques useful in the transition of modern software technology. What we have looked at is the technology maturation process, the natural process by which a piece of technology is first conceived and shaped into something usable and then "marketed" to the point that it is found in the repertoire of a vast majority of professionals.

A major interest is the time required for technology maturation--noting what time is required for various activities provides some baseline data with which to measure the effect of our transition strategies. But our prime interest is in finding out what actions, if any, could accelerate the maturation of technology, in particular that part of maturation that has to do with transitioning the technology into widespread use.

Technology transition activities are the heart of any technology improvement program. These are planned, overt actions taken to move a piece of technology into wide-spread use. They involve primarily the packaging of the technology so that it is well-received, the active insertion of the technology into some initial, high-impact arenas, and the ultimate dissemination of the technology to a broad community.

The details of our investigation (1-13) are available in Appendix G. The study of knowledge or technology maturation is a complex subject in itself (e.g., 14, 15). The basic technique we used to avoid some of the more obvious known pitfalls was to use persons highly knowledgeable in the technologies and their use to develop the case studies and summarize them.

In this report, we provide some general observations on software technology maturation, the basic context needed for transitioning technology into widespread use, and the activities that can inhibit or facilitate the widespread propagation of technology throughout the professional community.

B. TECHNOLOGIES INVESTIGATED

Fourteen software technologies were the focus of this study. They are:

- knowledge-based systems
- software engineering
- formal verification technology
- compiler construction technology
- metrics
- abstract data types
- structured programming
- SCR (Software Cost Reduction) methodology
- DoD-STD-SDS (Department of Defense-Standard-Software Development Standard)
- AFR 800-14 (Air Force Regulation)
- cost models
- Smalltalk-80
- SREM (Software Requirements Engineering Methodology)
- Unix

Some of these are very specific instances of a technology, such as the Smalltalk-80* system, and some are very broad technology areas, such as verification technology.

Four different types of software technology are represented in our sample. First, our sample includes the major technology areas of metrics, software engineering, compiler construction technology and verification technology. Advancements in major areas such as these depend on coordinated advancement in several interrelated areas, many of them frequently theoretical in nature. In addition, only a small segment of the technical community will directly use the technology in these areas. For example, few professionals actually generate compilers but virtually all of them use a (semi-automatically generated) compiler in their work.

The second type of technology that our sample includes is technology concepts such as abstract data types and structured programming. This type of technology is conceptual in nature and usually ends up as a basis for a variety of well-defined and usable pieces of technology. For example, structured programming led to the development of structured analysis.

Methodology technology is the third type found in our sample. This technology addresses how to develop and support software most effectively and is a mixture of technical and managerial principles, practices and procedures. This is basically "second-level" technology that provides the rules and guidelines

* Smalltalk-80 is a trademark of Xerox Corporation.

for how other technology can be best employed in the development and post-deployment support of software. The instances of this type of technology in our sample are: DOD-STD-SDS, the emerging Department of Defense (DoD) software life cycle model; AFR 800-14, the Air Force policy regarding the acquisition of embedded software systems; and the SCR methodology that underlies the Navy's Software Cost Reduction program.

Consolidated technology, the fourth type of technology found in our sample is also "second-level" technology. In this case, the underlying technology is collected and made to work together so as to provide something significantly better than any of the individual pieces of technology alone. A simple example is software cost estimation technology that brings together metrics, statistical prediction techniques, and empirical techniques. The other example within our sample is automated software development environments, for which we have considered three specific systems: Unix*, Smalltalk-80 and the Software Requirements Engineering Methodology (SREM).

C. TECHNOLOGY MATURATION

The fourteen individual case studies provide considerable detail about what happens during technology maturation. To provide some overall comparisons, these individual histories must be placed on a common scale and we use the one displayed in Figure IV-1. This figure defines six major phases for software technology maturation by fixing the time points that indicate passage between phases. Our interest in technology transition leads us to focus primarily on the period following the emergence of usable capabilities (time point 2).

*Unix is a trademark of AT&T Bell Laboratories.

SOFTWARE TECHNOLOGY MATURATION PHASES

```
                    * * * *  BASIC RESEARCH  * * * *
  -- investigation of ideas and concepts that later prove fundamental
     to the technology
  -- general recognition that a problem exists and discussion of its
     scope and nature

0 <======== Appearance of a Key Idea Underlying the Technology =====> 0
0 <============== or a Clear Articulation of the Problem ===========> 0

                  * * * *  CONCEPT FORMULATION  * * * *
  -- informal circulation of ideas
  -- convergence on a compatible set of ideas
  -- general publication of solutions to parts of the problem

1 <============ Clear Definition of Solution Approach Via a ========> 1
1 <============== Seminal Paper or a Demonstration System ==========> 1

               * * * *  DEVELOPMENT and EXTENSION  * * * *
  -- trial, preliminary use of the technology
  -- clarification of the underlying ideas
  -- extension of the general approach to a broader solution

2 <============== Usable Capabilities Become Available =============> 2

         * * * *  ENHANCEMENT and EXPLORATION (Internal)  * * * *
  -- major extension of the general approach to alternative problem
     domains
  -- use of the technology to solve real problems
  -- stabilization and porting of the technology
  -- development of training materials
  -- derivations of results indicating value

3 <========== Shift to Usage Outside of Development Group ==========> 3

         * * * *  ENHANCEMENT and EXPLORATION (External)  * * * *
  Same activities as for ENHANCEMENT and EXPLORATION (Internal) but
  they are carried out by a broader group including people who have
  not been involved in the technology maturation up to this point.

4 <======== Substantial Evidence of Value and Applicability ========> 4

                    * * * *  POPULARIZATION  * * * *
  -- appearance of production-quality, supported versions
  -- commercialization and marketing of the technology
  -- propagation of the technology throughout a receptive community of
     users
       a -- throughout 40% of the community
       b -- throughout 70% of the community
```

Figure IV-1

For all types of technology except consolidated technology, this is the point at which overt, planned actions can start to have some definite impact on the maturation of technology. In the case of consolidated technology, the process of fitting individual pieces of technology together may require some basic concept formulation and development and extension, and these preliminary activities may be amenable to acceleration.

The major time points for the individual technologies in our sample are indicated in Figure IV-2 and used in the graphs of technology maturation in Figure IV-3. Each time line in Figure IV-3 ends at 1984 and the time lines are adjusted so that the points for the appearance of a clear solution definition (time point 1) are lined up. The time lines are organized first by type of technology and second by the length of the development and extension phase. Since our interest is primarily in the period after usable capabilities become available (time point 2), the time lines are presented again in Figure IV-4 for those technologies that have achieved some degree of usage outside of their developing group, adjusted to emphasize the phases following the availability of usable capabilities (time point 2).

D. GENERAL OBSERVATIONS

Figures IV-3 and IV-4 indicate a wide variance in the time that it takes for a technology to mature from the emergence of a key idea to the point that it is used by professionals outside of the developing group or by the technical community at large. Nonetheless, the figures do indicate some general characteristics of the technology maturation process.

SOFTWARE TECHNOLOGY MATURATION POINTS

```
0     1     2     3     4
|     |     |     |     |
|     |     |     |     V
|     |     |     V     Substantial Evidence of Value and Applicability
|     |     V     Shift to Usage Outside of Development Group
|     V     Usable Capabilities Available
V     Definition Via Seminal Paper or Demonstration System
Emergence of Key Idea
```

Knowledge-based Systems
- 0 — 1965: appearance of artificial intelligence systems that provide intelligent assistance (for example, Dendral)
- 1 — 1973: appearance of systems containing a knowledge base (for example, Hearsay)
- 2 — 1978-80: appearance of knowledge-based systems that can be routinely used for problem-solving tasks (for example, R1)

Software Engineering
- 0 — 1960: inadequacy of existing techniques for large-scale software development noted in several projects (for example SAGE)
- 1 — 1968: concept of software engineering is articulated at Workshop on Software Engineering at Garmisch Partenkirchen
- 2 — 1973-74: general collections of papers appear and policy guidelines are established in various communities
- 3 — 1978-79: texts and generally usable systems supporting software engineering appear (for example, the SREM system)
- 4 — 1983: use of software engineering shifts to a larger community through actions such as the DeLauer directive and the definition of a Software Engineering Institute

Verification Technology
- 0 — 1966: Floyd's paper on program correctness analysis
- 1 — 1971: King's demonstration system appears
- 2 — 1975: multiple systems are available
- 3 -- 1979: usage of some systems shifts to application groups

Compiler Construction Technology
- 0 — 1961: Iron's paper on compiler generation
- 1 — 1967: review paper by Feldman and Gries
- 2 — 1970: usable systems appear (such as the XPL system at Stanford)
- 3 (cannot be determined)
- 4 -- 1980: appearance of production-quality compiler-compiler

Metrics
- 0 — 1972: publication of book on Halstead metrics
- 1 -- 1977: results of trying to measure various empirical and analytic measures appear

Figure IV-2

SOFTWARE TECHNOLOGY MATURATION POINTS (Continued)

Abstract Data Types
- 0 — 1968: initial report on information hiding
- 1 — 1973: appearance of some languages using idea of abstract data types (for example, TOPD design language)
- 2 — 1977: major publication on the subject and frequent appearance of the concept in new programming languages (for example, CLU)
- 3 — 1980: use of abstract data types in other technologies (such as in the Affirm program verification system)

Structured Programming
- 0 — 1965: Dijkstra's paper on programming as a human activity
- 1 — 1969: paper on structured programming by Dijkstra at the First Nato-sponsored Workshop on Software Engineering
- 2 — 1972-73: concept is widely discussed and presented in papers
- 3 (cannot be determined)
- 4 — 1976: publication of first introductory text based on structured programming

SCR Methodology
- 0 — 1968: appearance of concepts such as information hiding and communicating sequential processes
- 1 — 1976: completion of feasibility demonstration by NRL with positive experiences
- 2 — 1978-79: appearance of training material and models of usage
- 3 — 1982: methodology moved to a variety of other organizations

DOD-STD-SDS
- 0 — 1967: initial articulation of phased approaches to software development
- 1 — 1980: contract signed for development of DOD-STD-SDS

AFR 800-14
- 0 — 1972: basic need for policy and specific guidance is documented
- 1 — 1973: strawman policy is published
- 2 — 1974: policy guidance is published
- 3 — 1974: final draft is available
- 4 — 1975: regulation and instructions for its use are officially published

Cost Models
- 0 — 1966: appearance of first collection of cost-related data
- 1 — 1976: appearance of first usable system (Price S)
- 2 — 1978: alternative systems are available (for example, COCOMO)
- 3 — 1981: publication of Boehm's text

Figure IV-2

Smalltalk-80
0 — 1965: Kay's thesis defines concept of a personal computerized notebook
1 — 1972: preliminary version of Smalltalk is available
2 — 1976: major new version of Smalltalk appears
3 — 1981: other companies start porting the Smalltalk-80 system to their computers
4 — 1983: Smalltalk-80 available as a commercial product

SREM
0 — 1968: ISDOS system demonstrates applicability of attribute-value-relation approach to pre-implementation activities
1 — 1973-74: first concrete definition of the SREM system appears
2 — 1977: first release of the SREM system
3 — 1981: Vax version available

Unix
0 — 1967: appearance of the Multics system
1 — 1971: initial versions of Unix available
2 — 1973: Unix system debuts at Sigops conference
3 — 1976: collection of papers appears and system is beginning to be widely used in academic community
4 — 1981: announcement of Unix System III

Figure IV-2

SOFTWARE TECHNOLOGY MATURATION TIME LINES

```
Metrics
                    ... * * 0 * * * * 1 * * * * * * *
Knowledge-based Systems
              ... * * 0 * * * * * * * 1 * * * * * * 2 * * * *
Software Engineering
              ... * * 0 * * * * * * * 1 * * * * * 2 * * * * 3 * * * 4 *
Verification Technology
                    ... * * 0 * * * * 1 * * * 2 * * * 3 * * * * *
Compiler Construction Technology
                  ... * * 0 * * * * * 1 * * 2 * * * * * * * * * 4 * * * *

Abstract Data Types
                    ... * * 0 * * * * 1 * * * 2 * * 3 * * * *
Structured Programming
                      ... * * 0 * * * 1 * * * 2 * * 4 a * * b * * * *

SCR Methodology
              ... * * 0 * * * * * * * 1 * * 2 * * 3 * *
DOD-STD-SDS
    ... * * 0 * * * * * * * * * * * * 1 * * 2 *
AFR 800-14
                            ... * * 0 1 2 4 * * * * * * * * * * *
                                        3
Smalltalk-80
                ... * * 0 * * * * * * 1 * * * 2 * * * * 3 * 4 *
SREM
                  ... * * 0 * * * * * 1 * * 2 * * * 3 * * *
Cost Models
          ... * * 0 * * * * * * * * * 1 * 2 * * 3 * * *
Unix
                      ... * * 0 * * * 1 * 2 * * 3 * * * * 4 * * *


0             1             2             3             4
| Concept     | Development | Enhancement | Enhancement | Popularization
| Formulation |      &      |      &      |      &      |  a - 40% usage
|             | Extension   | Exploration | Exploration |  b - 70% usage
|             |             | (Internal)  | (External)  |
|             |             |             |             V
|             |             |             |             Substantial
|             |             |             |             Evidence of Value
|             |             |             V             and Applicability
|             |             |             Shift to Usage Outside of
|             |             V             Development Group
|             V             Usable Capabilities Available
V             Definition Via Seminal Paper or Demonstration System
Emergence of Key Idea

  Note:  the last point on each time line is 1984
```

Figure IV-3

SOFTWARE TECHNOLOGY TRANSITION PERIOD

```
Software Engineering
      ... * * 0 * * * * * * * * 1 * * * * * 2 * * * * 3 * * * 4 *
Verification Technology
               ... * * 0 * * * * * 1 * * * 2 * * * 3 * * * * * *
Compiler Construction Technology
               ... * * 0 * * * * * 1 * * 2 * * * * * * * * * * 4 * * * *

Abstract Data Types
               ... * * 0 * * * * * 1 * * * 2 * * 3 * * * *
Structured Programming
                ... * * 0 * * * 1 * * * 2 * * 4 a * * b * * * *

SCR Methodology
          ... * * 0 * * * * * * * * * 1 * * 2 * * 3 * *
AFR 800-14
                             ... * * 0 1 2 4 * * * * * * * * * * * *
                                         3
Smalltalk-80
          ... * * 0 * * * * * * * * 1 * * * 2 * * * * * 3 * 4 *
SREM
               ... * * 0 * * * * * * 1 * * 2 * * * 3 * * *
Cost Models
            ... * * 0 * * * * * * * * * * * 1 * 2 * * 3 * * *
Unix
                   ... * * 0 * * * 1 * 2 * * 3 * * * * * 4 * * *


0              1              2              3              4
|  Concept     | Development  | Enhancement  | Enhancement  | Popularization
|  Formulation |      &       |      &       |      &       |  a - 40% usage
|              |  Extension   | Exploration  | Exploration  |  b - 70% usage
|              |              | (Internal)   | (External)   |
|              |              |              |              V
|              |              |              |              Substantial
|              |              |              |              Evidence of Value
|              |              |              V              and Applicability
|              |              |              Shift to Usage Outside of
|              |              V              Development Group
|              V              Usable Capabilities Available
V              Definition Via Seminal Paper or Demonstration System
Emergence of Key Idea

  Note:  the last point on each time line is 1984
```

Figure IV-4

1. <u>Major Technology Areas</u>

For this type of technology, maturation requires a very long period of time. This is presumably caused by two things. First, since the area is broad, many advancements in specific pieces of technology are needed before a general advancement can take place in the area as a whole. Second, because the technologies we investigated in this area are ones that do not move into widespread use because of their specialized nature, full maturation must await the "pulling" effect of a recognized need for the product technology derived from the technology within the major area.

For several of the major technology areas investigated, maturation was guided by an external force. Verification technology was essentially languishing in the development and extension phase until its value for life-critical and secure application areas was recognized and special attention and focus was directed toward application areas with these characteristics. Compiler construction technology experienced a similar phenomenon when the general focus on high-level languages in the late 1970's forced attention to capitalizing on the technology that had been developed so far and developing the capability to produce economically production-quality optimizing compilers. Through the government's Ada and STARS programs, the external focusing force is starting to appear for the area of software engineering.

2. <u>Technology Concepts</u>

The two examples considered in this area share the characteristic that they are simple ideas with a fairly easily understood conceptual basis. Thus they have matured fairly rapidly; but they themselves have not spread throughout the technical community as fast as some of the technology based on them. For example, structured design, which is based on the concept of structured programming, is probably used by a larger segment of the technical community than is structured programming itself. This is to be expected--ideas can mature fairly quickly but it takes their unambiguous definition as a specific technique, perhaps supported by tools, to make them useable by the majority of the technical community.

3. Methodology Technology

Methodology technology concerns the rules and guidelines guiding use of other technology for the creation and evolution of software systems. As such, several things have to happen before this type of methodology can transition into widespread use. First, the underlying technology has to mature. Second, the rules and guidelines have to be developed. Finally, demonstrations of value have to be prepared.

The concept formulation phase for methodology technology can, therefore, be rather long as evidenced by the time line for DOD-STD-SDS -- the technology supporting full life cycle development methods just was not in place for this technology to mature faster. Once the technology is in place, then maturation can occur fairly fast as evidenced by the time line for AFR 800-14. But the definition and standardization process does not necessarily go quickly -- in the case of AFR 800-14 there was a relatively small, homogeneous community but in the case of DOD-STD-SDS, the community is much more diverse and the process of gaining approval is considerably slower. The value of good planning and the careful development of convincing cost/benefit demonstrations is shown by the experience in maturing the SCR methodology -- this project sets a standard for work in this area and indicates what can be expected in the "average" case that the methodology technology is well-developed and carefully transitioned into the professional community.

4. Consolidated Technology

The situation is similar for consolidated technology --many things have to come together in order for the technology to fully mature. From the case studies it appears that the enhancement and exploration phases take longer than for methodology technology, presumably because of the need to build the "glue" that fits various pieces of technology together. In fact, the phasing seems rather the same as for technology concepts but one must realize that consolidated technology must, of necessity, lag the maturation of the technology on which it is based.

E. ACCELERATION OF TECHNOLOGY MATURATION

We have too few case studies and the areas are too disparate to be able to determine the nominal case for technology maturation. In fact, one suspects that special considerations will make each instance rather unique and that it will be hard, if not impossible, to predict the maturation time line for a technology by investigating the time lines for other technologies even if they are quite similar.

Our case studies do, however, indicate a number of factors that can inhibit or facilitate the maturation of technology. These give us some insight into what we should and should not do in order to assure that technolcgy matures as smoothly as possible. These factors are discussed in this section.

A word of caution: the various factors are discussed individually but they will interact in very complex ways for any given technology and we do not address these interactions here. An interesting study that reflects the overall effect of all factors impinging on the maturation of technology has been done by Graham (16) who notes that there has been a 50-year cycle in the popularization of technology and makes some interesting observations as to why this is so and how it will affect the transition of software technology.

1. Critical Factors

The case studies indicate a number of factors that are critically necessary in the sense that trying to move a technology into widespread use is almost pointless unless these factors are present. Our case studies show a few instances of failure when these factors have not been present. In many cases, however, total failure was avoided by realizing the problem and correcting the situation -- therefore, many of our examples in this section really show the slowing effect of a failure to establish the right context for technology transition.

Conceptual Integrity. The technology (or the base technology in the case of methodology or consolidated technology) must be well developed. In particular, there can be no major outstanding questions about the conceptual basis underlying the technology -- the resulting controversy will just slow things too much. The area of metrics indicates how controversy can make any meaningful progress impossible. On the other hand, the conceptual clarity of the Unix operating system (due in part to the preceding work on Multics) made its maturation possible. As another positive example, the clean separation of concerns within the area of compiler construction technology has led to that area being one of the few major technology areas that has matured to popularization.

Clear Recognition of Need. The technology must fill a well-defined and well-recognized need. Often this need just materializes, but in many cases the need must be articulated by a well-respected salesperson. Time points 1 and 2 in our maturation process reflect the critical role played by clear enunciation of a need and a solution.

Tunability. It must be possible to mold and tune the technology to the specific practices of a variety of technology user groups. Both SREM and Unix provide examples of highly tunable technologies that could be molded to a variety of "ways of doing business".

The transparency of the COCOMO (Constructive Cost Model) cost estimation model is another example of how a technology can be open to modification and therefore more easily incorporated into diverse situations.

Prior Positive Experience. Reports on prior positive experiences with the technology should be readily available. Of particular importance are reports showing demonstrable cost/benefit. This is evidently a major contributing factor in the successful transitioning of the SCR methodology. The success of the Price-H hardware cost estimation model was evidently a primary factor in the Price-S software cost estimation model being almost immediately accepted and used. On the other hand, the lack of demonstrable success in applying SREM to software development caused its maturation to be considerably slowed. (The SREM experiences demonstrate a severe problem that will exist for much of the technology surrounding early life cycle phases --clear demonstrations of value must await the completion of a reasonable sized project and this will delay the results.)

Management Commitment. Management must be committed to the introduction of new technology. And this means that they must actively work to introduce the technology rather than just not oppose its introduction. The case study of the SCR methodology cites at least one case in which the technology introduction failed because of a lack of management commitment.

Training. Training in the use of the technology must be provided and this training should include a large number of examples. This training is particularly critical when new, modern concepts are involved since then the users must be put in the "right frame of mind" before they can effectively use the technology. The studies of SREM, the SCR methodology, and verification technology all cite the criticality of high-quality training.

2. Inhibitors

The factors discussed in the last section are critical and if they are not present then the situation must be corrected. Even when they are present, and therefore maturation can proceed, the case studies show that a number of inhibiting factors can slow down the process (as opposed to bringing it to a standstill).

Internal Transfer. It may take additional time to propagate a technology throughout an organization. Our own case studies do not uncover the need for this additional time, but it is demonstrated by a particularly complete and fairly quantitative study of technology transition that was done by Willis (17). He notes two important factors that affect the internal propagation of a technology but do not show up in our case studies. First, he notes that the influence of new hires, who are knowledgeable in the new technology and less reluctant to use it, can be important in facilitating the transition of technology. Second, he notes the general requirement that there be some person or group of persons who are personally committed to successfully transitioning the technology -- this is consistent with Boehm's suggestion that there be a technology transition agent within companies to aid and guide the infusion of new technology (18).

High Cost. The cost, either in money or in the time needed to grasp the technology, must be reasonable. The cost estimation technology study cites the high monetary cost of Price-S as inhibiting its acceptance and the intellectual difficulty of COCOMO as having a similar effect. The study of SREM relates the SREM developers' belief that the cost of using the initial version, although reasonable, was above a preconceived threshold and that this delayed the acceptance of SREM. And certainly, the difficulty of performing verification, even with automated aids, has slowed its transition into general use. (One would expect that a high cost would be all right as long as the derived benefit was high, leading to a high benefit-to-"pain" ratio. Our cost studies do not support this; rather they indicate that the cost must be reasonably low whatever the benefit gained.)

Contracting Disincentives. Acquisition and contracting practices can serve to slow the spread of technology. The case study of SREM points out that private industry may be reluctant to support the development of new technology when the result will be in the public domain and the developing organization will not gain a competitive edge as a result of their efforts. And the SCR case study mentions the possibility that modern technology for developing maintainable systems will not be used when the possible result will be the unavailability of lucrative follow-on work.

Psychological Hurdles. Many practitioners feel threatened by new technology, especially when it is advertised as changing (or worse, automating) processes that they have been competently doing for years. And the computer science community seems particularly afflicted by the "Not Invented Here" disease that makes practitioners think that they have developed or can develop something much better than what is being offered for use.

Easily Modified Technology. If something can be changed or fiddled with, then it is almost axiomatic that a computer scientist will change it or fiddle with it. Therefore, if a technology is easily modified (as opposed to just tuned), its introduction will be slowed because it will be modified. This was cited as a debilitating factor in at least one case of introducing the SCR methodology.

3. Facilitators

Technology will spread more quickly when the inhibiting factors mentioned in the previous section are absent. But, our case studies indicate that there are also a number of factors that can tend to speed the dissemination of technology.

Prior Success. A "good track record" for the technology's originator(s) will not only make it easier to "sell" but may lead to practitioners seeking out a technology when they read or otherwise hear about a recognized expert's new developments.

Incentives. Software acquisition contracts can specify that new technology must be used -- for example, the Ballistic Missile Defense Advanced Technology Center is currently requiring the use of SREM on some of its contracts. In some cases, the incentive can be indirect -- contract bidders were evidently quite interested in using particular cost estimation models when it was learned that the government was using these models in their proposal evaluation.

Technically Astute Managers. In two case studies, it was noted that adoption of a new technology went more quickly when the decision-makers were well versed in modern software technology.

Readily Available Help. Knowledgeable, articulate advisors and consultants will help in explaining a complex technology, in getting over the almost inevitable misunderstandings that will arise and in dispelling any initial hostility. They can also serve indirectly as salespersons who can make needs visible, explain the benefits of the new technology, and relate the technology to the needs of potential users.

Latent Demand. If there happens to be a well-recognized critical need for the technology, then its adoption can be almost immediate -- this was experienced in the case of cost estimation models where the need was a longstanding one and the early models were almost immediately used.

Simplicity. While the technology and its underlying basis can be quite complex, adoption will move more certainly and smoothly if the instances of it that are available for use are easy to comprehend and are only minimally disruptive to the state of practice. The slowness in maturing verification and knowledge-based system technology is partially due to the absence of simple, easily comprehended systems that deliver this technology. On the other hand, the relative simplicity and conceptual clarity of the Unix operating system contributed to its rather quick adoption by the academic and research community.

Incremental Extensions to Current Technology. This factor is closely related to simplicity. Technology that requires large cognitive switches, such as the SREM and verification technology, will transition relatively slowly whereas technology that is an incremental enhancement of previous technology, such as the Unix system, will be adopted rather quickly.

F. CONCLUSIONS

The case studies show that it takes on the order of 15 to 20 years to mature a technology to the point that it can be popularized and disseminated to the technical community at large. (And other studies (16) show that it may take another four to eight years for the technology to transition within various organizations.) In special cases, this time can be considerably shorter. One such case is technological concepts and ideas that do not need computerized support or a change in the using community's mind set. Another special case is methodology technology that can transition very quickly when the need is clear and the target community is a close knit and homogeneous one.

The case studies show a very mixed situation with respect to government coordination as a means of facilitating technology transition. AFR 800-14 was established very quickly, mainly because of the homogeneity of the target community. DOD-STD-SDS, on the other hand has gone through at least one major revision because of the difficulty of arriving at a consensus among the technical community -- this has considerably slowed its maturation. Even in the case that there is a clear need and a strong desire to agree on a piece of technology that can be used throughout the government, the decision-making process can slow the maturation considerably -- the need for a common high-level language within the DoD was articulated as early as 1971 (18) but the initial definition of Ada was not chosen until eight years later and it took another four years to finalize the language definition support.

No technology will transition into widespread use unless there is a recognized need, a receptive target community and believable demonstrations of cost/benefit. In addition, bringing a technology into widespread use requires a well-defined channeling of attention accompanied by concentrated support. It also needs an articulate advocate who will argue both for the need to support development and the value of the technology once it is developed.

The best process for transitioning technology seems to be incremental expansion in small steps with trial use and the careful gathering of empirical evidence concerning the technology's value. The transitioning of Unix is the epitome of this approach. Unix was initially an incremental improvement over Multics and the Unix system appeared as a series of systems, each being an incremental improvement over the previous instance. Government constraints upon AT&T contributed to the failure to commercialize Unix and this slowed what might have been a relatively quick transition. The carefully managed exploration of Smalltalk-80 by professionals outside the development group is a good example of how the initial part of technology transition can be done in controlled, small steps.

Compiler construction technology provides a slightly different example of this process. Early in the development of compilers there was a clean separation of the problems into a number of inter-related concerns: parsing, optimization, code generation, etc. This separation made it possible to make improvements that were relatively small with respect to the area as a whole and that could be relatively easily delivered to the compiler development community.

For all of the cases we looked at, technology transition was inhibited not by making major conceptual or strategic blunders, but rather by making small, relatively simple mistakes that were easy to correct once they were identified. SREM provides a good example. With 20-20 hindsight, it was obviously a mistake to implement SREM initially on a super-computer. The decision was a reasonable one because of the risk associated with pre-implementation support and the availability of the TI ASC computer to support the project. But the restrictions that this imposed on the target user community and the difficulty in porting that resulted caused a significant slowing in the transitioning of this technology. Once the problem was appreciated and the resources obtained to fix the situation, SREM was ported to a much more widely available host in less than a year and its transition to wider use put back on track.

Our case studies also show that, in general, technology transition is facilitated by actions that improve the context in which the transition is taking place rather than address the technology itself. Of course, research and development is needed to assure that the technology is sound and complete. But this work does not seem to be easily accelerated. Rather, it seems that acceleration will come more from improving the context through actions such as providing a focusing goal, improving the technical capabilities of the target community, assuring that training materials and personnel are available, and assuring that the technology "packages" that are provided for use are conceptually coherent and relatively simple improvements over the technology already in use.

Our study shows that a number of factors can affect the speed at which technology matures and is transitioned into widespread use. Even without the inhibiting effect of not providing the basic ext or making mistakes, the technology will take a long time .ature. The degree to which the maturation can be speeded seems limited but there are several actions, relating to the context in which the technology is maturing, that can be taken to accelerate technology maturation.

REFERENCES

(1) John Bailey. Cost Model Technology Transition. May 1984.

(2) Paul C. Clements, et al. Case Studies of Software Engineering Technology Transfer. Tech. Memorandum, Naval Research Laboratory, April 1984.

(3) Richard A. DeMillo. Compiler Technology Insertion Network Study. May 1984.

(4) John H. Manley. Technology Case Study: Software Engineering Concepts. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, May 1984.

(5) John H. Manley. Technology Case Study: Software Metrics. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, April 1984.

(6) John H. Manley. AFR 800-14 History. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, May 1984.

(7) Ann Marmor-Squires. Formal Software Verification as an Example of Software Technology Transfer. TRW Defense Systems Group, Fairfax, VA, May 1984.

(8) Ronnie J. Martin. DOD-STD-SDS: The Development of a Standard. May 1984.

(9) Samuel T. Redwine, Jr. Structured Programming: A Technology Insertion Case Study. Computer and Software Engineering Division, Institute for Defense Analyses, May 1984.

(10) William E. Riddle. "The Magic Number Eighteen Plus or Minus Three: A Study of Software Technology Maturation." ACM SIGSOFT Software Engineering Notes, 9, 2 (April 1984). (Includes case studies of Unix, Smalltalk-80, and SREM.)

(11) William E. Riddle. Knowledge-based Systems as a Case Study in Software Technology Maturation. SDAM/15, software design & analysis, inc., April 1984.

(12) William E. Riddle. Abstract Data Types as a Case Study in Software Technology Maturation. SDAM/16, software design & analysis, inc., April 1984.

(13) David Weiss. Time Line for Development and Transfer of SCR Methodology. February 1984.

(14) R.F. Rich (editor), The Knowledge Cycle, Sage Publications, 1981.

(15) William D. Garvey. Communication: The Essence of Science, Pergamon Press, 1979.

(16) Alan K. Graham. "Software Design: Breaking the Bottleneck." IEEE Spectrum, March 1982, pp. 44-50.

(17) R. R. Willis. "Technology Transfer Takes 6 Plus/Minus 2 Years." Proc. IEEE Workshop on Software Engineering Technology Transfer, April 1983, Miami, Florida.

(18) Barry W. Boehm. "Keeping a Lid on Software Costs." Computerworld, January 18, 1982.

(19) Paul Cohen. Early Software Technology Efforts Within DCA. March 1984.

CHAPTER V STATE OF THE ART

A. INTRODUCTION

The state of the art in software technology has been addressed by a series of studies in recent years (1)-(5). In addition to these broad ranging studies, a number of surveys in more narrowly defined fields have appeared, for example, in artificial intelligence (6)-(7). Rather than provide another survey, the intent of this chapter is to give concrete examples of the state of the art in software technology that help provide insight into the potential for software technology improvement and address potential benefits. A detailed plan for a software technology improvement program requires a detailed assessment of the state of the art (e.g., 5). But, the assessment provided here is intended to support high-level, strategic planning and provide concrete examples. Rather than provide an exhaustive accounting, it therefore focuses on a representative sampling of current technology.

Our primary interest is in technology that supports the development and maintenance of software systems. We have chosen eleven technology topic areas that span the variety of techniques and tools providing a workspace for software development and maintenance. In the general area of techniques, we have chosen the following topic areas:

-- development methods: sets of rules and guidelines that serve to discipline the process of software system development,

-- testing technology: techniques for assessing a system's validity that require system execution,

-- static analysis techniques: techniques for validity assessment that rely on formal analysis rather than execution,

-- verification techniques: techniques for assessing whether the system will meet its specification (as opposed to the users' expectations).

These topic areas emphasize the critically important activity of analyzing the suitability of a system under development or enhancement. In the general area of tools, the topic areas are:

-- program transformers: tools that transform a program's text into an executable version,

-- pre-implementation modelling notations: languages for rigorous but abstract description of a system during the definition of its requirements or the development of its design,

-- measurement technology: metrics and experimental paradigms supporting the assessment of system quality

-- compiler generation technology: tools that automatically generate part or all of another tool, in this case a compiler,

-- software engineering environments: collections of tools that support all or part of the software life cycle,

-- editors: the tools provided for entering and manipulating text,

-- command languages: languages for controlling the invocation of tools within a software engineering environment.

In this set of topics we have tried to treat not only the tools themselves but also tools that aid the development of tools as well as several aspects of tool collections.

Our interest is in technology that has passed from the research arena and been matured into a usable "product" but for some reason is not widely used* within the practitioner community.

*We are not interested here in analyzing why the usage is not wide-spread. Some observations are made in this regard, but only as needed to argue that our categorization of various tools and techniques is appropriate.

For each of the topic areas, Figure V-1 indicates four techniques or tools that help understand what lies within this area of interest. We bound the area of interest for each topic area by indicating 1) a tool or technique that has made it into the repertoire of the general practitioner and 2) a tool or technique that is currently still in the research phase. Within these bounds, we cite two state-of-the-art tools or techniques. First, we indicate a tool or technique that is on the verge of emerging into more wide-spread usage. Second, we indicate a tool or technique that, although well-researched and well-developed, has only a very small user community. Thus, for each technology topic area, we provide examples at each of the following levels of transfer into wide-spread usage:

-- mature: there are several different implementations available and these are used by a broad segment of the practitioner community,

-- emerging: the tool or technique is generally available but not widely used, perhaps because of a lack of a variety of implementations or maybe because of a lack of "enthusiasm" within the practitioner community even though a variety of implementations are available,

-- understood: the tool or technique is well-researched but the available implementations (if, in fact, there is more than one) are used by only a small segment of the community, primarily only the tool's or technique's developer(s),

-- research: the concepts underlying the tool or technique are still being researched.

Summary of State of The Art of Software Technologies

STATE OF THE ART	**MATURE** WIDELY-USED	**EMERGING** GENERALLY AVAILABLE BUT NOT WIDELY USED	**UNDERSTOOD** WELL-RESEARCHED BUT NOT WIDELY AVAILABLE	**RESEARCH** UNDERLYING CONCEPTS UNDERGOING RESEARCH
METHODS	FUNCTIONAL DECOMPOSITION	DATA DECOMPOSITION	FULL LIFE CYCLE METHODS	NON-WORK-PRODUCT METHODS
TESTING TECHNOLOGY	AD HOC TESTING	COVERAGE TECHNIQUES	FUNCTIONAL TESTING	TEST CASE SELECTION TECHNOLOGY
STATIC ANALYSIS TECHNIQUES	SYNTAX CHECKING	TYPE CHECKING	DATA FLOW ANALYSIS	USER-STATED CHARACTERISTICS
VERIFICATION TECHNIQUES	WALK THROUGHS	RIGOROUS "DESK-CHECKING"	VERIFICATION SYSTEM	TRANSFORMA-TIONAL DEVELOPMENT
PROGRAM TRANSFORMERS	COMPILERS	PROGRAM BEAUTIFIERS	CONVERSION AIDS	PROGRAM GENERATORS
PRE-IMPLEMEN-TATION MODELING NOTATIONS	PROGRAM DESIGN LANGUAGES	ABSTRACT DATA TYPES	DESIGN MODELING NOTATIONS	RE-USABLE SOFTWARE PARTS
MEASUREMENT TECHNOLOGY	SIZE METRICS	DATA COLLECTION	PROGRAM METRICS	EXPERIMENTAL TECHNIQUES
COMPILER GENERATION TECHNOLOGY	PARSER GENERATORS	LEXICAL ANALYZER GENERATORS	PRODUCTION QUALITY COMPILERS	RETARGETABLE COMPILERS
SOFTWARE ENGINEERING ENVIRONMENTS	PROGRAMMING SUPPORT SYSTEMS	REQUIREMENTS DEFINITION SYSTEMS	LIFE CYCLE SUPPORT ENVIRONMENTS	KNOWLEDGE-BASED ENVIRONMENTS
EDITORS	FULL-SCREEN EDITORS	INTERACTIVE EDITORS	SYNTAX-DIRECTED EDITORS	GRAPHICS-BASED EDITORS
COMMAND LANGUAGES	JOB CONTROL LANGUAGES	PROGRAMMABLE COMMAND LANGUAGES	GENERIC COMMAND LANGUAGES	TOOL AGGREGATION

Figure V-1

B. DISCUSSION

1. Methods

The development and maintenance of software systems, particularly large-scale or complex ones, requires discipline. Methods provide the rules and guidelines that impose this discipline both on individual practitioners and upon teams working together on a single system.

Functional decomposition methods focus attention on the capabilities to be delivered by the software system and guide the hierarchical decomposition of this functionality into gradually more primitive functions. Structured programming (8) is an early instance of such a method which relies on the use of a small number of understandable control constructs to detail the way in which sub-function invocation is controlled. More recent methods of this sort, for example structured design (9), focus less on the invocation control mechanisms and more on the logical decomposition of functionality and the definition of information flow among functional components.

Data decomposition methods switch attention from the processing to be done to the data that is the subject of processing*. The basic rationale is that in many situations, the decomposition of the data is more easily accomplished and that the processing to be done can be inferred from a data decomposition once it is defined. The most extensively developed method of this type is owed to Jackson (10).

* Data flow techniques help in the decomposition of functionality and therefore are considered to be in the class of functional decomposition methods.

Object-oriented design (11), which is founded on the concepts of abstract data types (12), is another method of this sort that is receiving increasing attention.

Current data decomposition methods generally focus on one phase of the software development process, usually the programming phase. Less well-developed are methods that span a variety of phases. The Jackson method has been extended to cover the design phase as well as the programming phase (13). The development of the Distributed Computing Design System (DCDS) (14) has resulted in an understanding of the methods supporting system requirements definition, software requirements definition, software architectural design, and software detailed design. This project has also resulted in an understanding of the methodological guidelines for moving among these phases. While methods such as these are not yet full life cycle methods, their existence does indicate that organizing and disciplining the full life cycle process is beginning to be well-understood.

Less well understood are non-work product methods such as prototyping (15). A concern for management control of the software development and maintenance process has driven the majority of methodology work. Current methods are therefore focused on, or at least have been aligned with, a collection of work products that provide management visibility and control. Prototyping, on the other hand, is a technique that focuses attention on the validation issue of whether or not a suitable system will result. Alternative types of methods and their compatibility with existing work product-oriented methods are currently a focus of active investigation.

2. Testing Technology

Validation of a system by making trial runs and checking the output, i.e., testing, is a long-standing practice. The generally used technique can be called ad hoc testing (16) because information concerning the system's overall function is used in a relatively ad hoc manner to determine test cases.

Currently emerging into more wide-spread use are coverage techniques (17) that additionally use information about a program's physical structure in guiding the selection of a set of test cases. For the most part, these techniques focus on assuring that a large proportion of the paths through a program's logic are exercised, or at least measuring the percentage of paths so that the user can understand the extent to which a program has been exercised. Several coverage testing tools are commercially available (18), (19), but the use of coverage testing techniques is not wide-spread.

Functional testing has been proposed (20) as a way of using information about not only the system's overall intended function but also the "internal" functions that are defined during design. The technique itself and its limitations are well-understood but more investigation is needed before it can be used by even a small segment of the practitioner community at large. In particular, it appears that a family of such techniques, each pertaining to a specific application area, will have to be developed (21).

The selection of test cases is currently done in a structured but essentially ad hoc manner. Some of the current testing research (as reported, for example, in (22) and (23)) is oriented toward developing a formal test case selection technology that provides a rigorous foundation for test case selection. A side effect of this research will be a firmer understanding of the characteristics of different testing approaches and an ability to compare, contrast and select testing strategies.

3. Static Analysis Techniques

An alternative to assessment by execution, such as in testing, is to analyze the text of a program for errors. This so-called "static" approach to assessment may involve some interpretation (conceptual execution) of the program and is therefore often similar to dynamic approaches such as testing. The distinguishing characteristic is the scope of the technique -- static analysis techniques check for all errors of a certain type whereas dynamic techniques, except in very restricted error domains, uncover only some of the errors of a certain type.

Every practitioner uses a high-level language (for at least part of the time) and thus uses a syntax checking static analyzer since this function is always performed by the high-level language's compiler.

Less common is the use of a static analyzer that performs type checking. Many modern high-level languages, such as Pascal and Ada, have strong typing rules that require the declaration of a type for every data object and conformance to strict type-related rules about the use of data objects. While there is some controversy about the value of such rules in developing some types of software (for example, system's software), the general feeling seems to be positive. The use of this type of static analyzer will increase as the use of languages with strong typing rules increases.

Static analyzers of the sort discussed basically check for organizational or structural errors. They can check for behavior-related errors only to the extent that these errors can be defined in organizational or structural terms. Data flow analyzers, on the other hand, directly check for behavioral errors that concern the usage of data values. For example, a typical check made by a data flow analyzer would be for the use of an undefined variable. Data flow analysis techniques for sequential programs are well-understood (24) but more engineering, such as being done in the Toolpack project (25), needs to be done before this class of static analyzers are both effective and efficient. Some work ((26), (27)) has been done on data flow analysis techniques for concurrent programs but more research is needed on making these techniques computationally attractive.

Static analysis techniques generally check for pre-defined errors, that is, errors that stem from the definition of the programming language (such as type-related errors) or from knowledge of "good practice" (such as a division-by-zero error). A current research issue is whether data flow analyzers can be more generic and accept definitions of the errors to be checked for. The result would be data flow analyzers that check for user stated characteristics where the user in this case is the builder of a software engineering environment who wants to include a data flow analyzer that checks for a certain type of error.

4. Verification Techniques

The static and dynamic analysis techniques discussed in the last two sub-sections are oriented towards software validation, that is, the assessment of whether or not the system will meet user desires. Verification techniques address a different issue, namely whether the system being constructed will meet the user's requirements as stated in a requirements specification. The concern in verification is less with whether the eventual system will prove acceptable under actual use and more with whether it is correct with respect to its stated requirements.

Walk-throughs (28) are typically used to verify a system's design or implementation. These consist of well-organized peer reviews during the development process. They are founded on the phenomenon that errors are best discovered by someone other than the person who developed the design or code. They have the side effect of causing the intra-team interactions critical to discovering interface errors.

Rigorous "desk-checking" takes the level of analysis one step farther. Whereas the analysis in a walk-through is primarily informal, the attempt in rigorous desk-checking is to analyze more formally the behavior of a program or software system. Sometimes this is done with just careful case analysis or hand tracing of representative cases. Some techniques have been developed to provide even more formality. For example, symbolic execution (29) can be used to perform rigorous desk checking. In symbolic execution, algebraic values are used to represent input values, rather than using specific numerical values as in testing. A primary value of symbolic execution is that it can be used to obtain information about the function that will actually be computed under certain domains of input values so that this can be (manually) checked against a perception of the function that is to be computed. A variety of implementations of symbolic execution analyzers are available, but usage is rather small.

In formal verification, a rigorous mathematical analysis is used to, (1) determine the function that a program will actually compute and, (2) compare that function with the function that the program is intended to compute. Formal verification technology is fairly well-developed for sequential programs and has resulted in several verification systems such as Affirm (30) and HDM (31). Both their restriction to sequential programs and the need for an extensive amount of training and education in the theory underlying formal verification hinder their wide-spread use.

The ultimate in verification is to prepare the designs and programs automatically by transforming one description into a more detailed description closer to the eventual implementation. For example, a technique has been developed for automatically choosing implementing data structures in several cases (32). The technical problem that arises in such transformational development techniques is that one must verify that the transformation itself is correct, that is, that its output will always be a correct elaboration of its input.

5. <u>Program Transformers</u>

With the introduction of high-level languages came the associated need to transform automatically a high-level language program into executable text. Compilers were developed as a tool to perform this transformation and they are widely used throughout the practitioner community.

The advent of a concern for well-structured programs brought with it, the need for transforming existing, poorly structured programs into well-structured programs or for transforming programs of any sort into ones that exhibited a standard structure. Program beautifiers, such as the struct tool (33) in Unix*, have been developed to meet this need but are not as yet widely used.

* Unix is a trademark of AT&T Bell Laboratories.

Conversion aids, such as reported in (34) and (35), are another type of program transformer. These convert programs in one high-level language into an equivalent program in another high-level language. Several conversion aids have been developed but their wide-spread use requires more knowledge about how to handle machine dependencies as well as techniques for assuring that the result is a well-formed and efficient program in the target language.

Program generators such as discussed in (36), are fairly well developed in specific, generally very narrow, application areas. Tools of this sort accept a definition of the processing required of a software system and automatically generate the system. Successful automatic generation requires a rather exact knowledge of the algorithms to be used in accomplishing the required processing. This knowledge is, at the moment, known only for relatively narrow application domains. More extensive usage requires some very basic research on how to generalize the experiences to date to provide a capability over a reasonable spectrum of applications.

6. Pre-implementation Modelling Notations

A program in a high-level language is a description of a software system that can be automatically transformed into an executable version. In a very real sense, it is a model of the executable version since it omits some of the excruciating detail that appears in the executable version --for example, the details of array address computation are omitted in high-level scientific programs written in Fortran. Pre-implementation modelling notations provide for additional levels of abstraction useful in describing software systems during the pre-implementation phases of design and requirements definition. Their main value is in allowing a rigorous description of (some aspects of) a software system before all of the detail has been determined.

Program Design Languages, such as initially described in (37) allow the abstract description of a program with a focus on the control logic and a de-emphasis of the data manipulation performed by the program. Use of this sort of design modelling notation is relatively wide-spread throughout the practitioner community.

Abstract data types (12) are another design-oriented modelling technique that allow the description of a module's functionality without the need to describe how the module achieves the described functionality. Most languages that provide abstract data type capabilities are in the research community, with the notable exception of Ada. The more wide-spread use of Ada and languages of its type will carry with it the more wide-spread use of the concept of abstract data types. In this regard, it is interesting to note the recently expanding use of Ada as a design, instead of a programming, language.

Several design modelling notations (such as found in the TOPD (38), Gypsy (39) HDM (31), and SARA (40) software engineering environments) have been developed to provide a means of describing a software system's modules, their functions, and their interactions. These notations allow the description of a software system's architectural design in which the system's overall structure is highlighted in a way that aids in subsequently working out the processing details of the individual modules in the structure.

Capturing the information needed for selecting existing modules for reuse in a new software system requires a different type of abstract description. In this case, the description must communicate use-related rather than implementation-related information -- for example, information concerning the performance of the module is much more pertinent in this case. The problems of providing libraries of reusable software parts, discussed in (41), are research-level problems at this point. (Current-day libraries of mathematical routines provide a good starting point for research into more general solutions.)

7. <u>Measurement Technology</u>

Measures of the process of software development and maintenance as well as the products produced by this process are critical to evaluating the efficacy of alternative approaches or techniques. The only measures that are widely used are size metrics, with the most extensive use of these measures being for cost estimation (42). Other metrics are not widely used because of a suspicion that they do not highly correlate with the real item of interest such as programmer productivity or program complexity.

A currently emerging technology in this topic area is the technology of data collection. While there is far from universal agreement on what data should be collected, there is the wide-spread realization that data must be collected in order to advance the general state of affairs as regards measurement technology. This realization has led to, among other things, the establishment of the Data Analysis Center for Software at Rome Air Defense Center that has the charter to provide a repository for software product - and process-related data and a mechanism for the dissemination of this data. The current state of affairs seems to be that the level of consciousness about the need for data collection is high enough that long-standing techniques, such as instrumentation, are being more widely used.

More complex program metrics have been developed in an attempt to provide a means of measuring a program's abstract characteristics (such as complexity) by measuring the program's concrete characteristics (such as the connectivity of a graph representing the program's control logic). While the theory underlying the metrics themselves is generally well-understood, the general lack of understanding of whether these metrics really measure the abstract characteristics of interest has inhibited their more wide-spread use. The metrics developed by Halstead (43) are a case in point -- they pretend to measure a program's understandability (and therefore several other "ilities" such as maintainability) by measuring characteristics such as the number of unique objects and operators, but there is far less than general acclaim that they do (44).

Even farther within the research arena in the area of measurement technology is the subject of experimental techniques. An experiment to evaluate comparatively alternative architectural design techniques with respect to their support for a system's maintainability has been proposed (45). Much more research needs to be done before software and the processes for developing and maintaining software can be experimentally evaluated in a truly scientific manner (46).

8. Compiler Generation Technology

A compiler generator is a piece of software that accepts a definition of the language to be compiled and the machine that is the target of the compilation and automatically produces the compiler. Such software is an example of tool-building tools that are not of direct utility to software development and maintenance practitioners but are rather of use in preparing the tools that these practitioners may use.

Techniques for automatically generating the parser segment of a compiler were well-known in the late 1960's (47). Very few compilers are built today without the use of a parser generator to automatically generate the compiler's parser.

Lexical analyzer generators are also available to automatically produce the segment of a compiler that splits the input stream into lexicographic tokens (i.e., words) for subsequent processing by the parser. Their use seems to be a bit less wide-spread, perhaps because the task of lexical analysis is both simple enough to be manually programmed and complex enough for many modern programming languages (because of dependencies on contextual information such as appears in variable definitions) that it is impossible to fully automatically generate a lexical analyzer. Lexical analyzer generators, such as Unix's lex (48), are generally available and are receiving more wide-spread use in the development of all sorts of tools since the task of lexical analysis recurs in many contexts.

A few production-quality compiler compilers, that attempt to provide support for all aspects of compiler development, have been developed. At least one, the PQCC system (49) developed by Wulf, has been the basis for launching a commercial enterprise. But, so far, the use of this extensive, coherent technology is rather limited across the community of compiler builders.

The "back-end" of a compiler, that part that produces the actual target system code, has only recently been addressed within the research arena (50) with attempts to provide retargetable compilers (as opposed to portable compilers). While there has been extensive success, more research is needed before this technology can be actively used by even a small segment of the compiler building community.

9. Software Engineering Environments

Collections of tools are more supportive of software development and maintenance than are individual tools. This is especially true when the collection supports a specific method or covers much or all of the software life cycle. Operating systems are a simple example of collections of tools, but these provide only minimal support for just the preparation of a software system's code. In the last decade, interest has become strong in going beyond operating systems to provide software engineering environments that support teams of practitioners over a fairly broad spectrum of the software life cycle.

A broad segment of the practitioner population uses programming support systems. For the most part, these are modern operating systems that support the preparation of an implementation in a high-level language, the debugging of the implementation and the maintenance of the large collection of modules that typically exist for a large-scale software system. More advanced programming support systems have been developed (for example, Unix (51), Interlisp (52), and Smalltalk (53), but these generally have only a small, albeit it very strong, user community.

Several requirements definition systems have been developed to provide automated support for the recording and analysis of software system requirements. Most notable are PSL/PSA (54) and SREM (55), each of which has a relatively extensive user population. These systems are starting to be used by choice rather than duress and it would seem that their further commercialization will lead to relatively wide-spread use.

The emerging understanding of how to cover the entire software life cycle with a set of compatible, integrated tools has led to the development of life cycle support systems. The previously mentioned DCDS system (14) is one such system. Other examples are the USE system (56) that supports the development of interactive information systems and the Joseph system (57) that supports pre-implementation activities during the development of multiple-processor systems software. All of these systems are, however, research-level systems that need considerable enhancement and the development of production-quality versions before they can be extensively used.

A focus of extensive research interest in the software engineering environments area today is the use of artificial intelligence techniques to provide knowledge-based environments ((58), (59)). Such environments would possess knowledge of various software development techniques and the conditions under which they should be applied. This knowledge would be automatically used to more or less automatically derive an executable version of a software system from the specification of the system's requirements. With such systems, development practitioners would no longer use the assistance of automated tools to develop software but rather would become general strategists who are "consulted" when experience, insight and intuition are needed during software development.

10. Editors

The vast majority of software practitioners access a software engineering environment interactively through terminals. Much of the use of these terminals is to prepare and manipulate the text of the software system under development through the use of editors. It is very common to have full screen editors, such as Unix's vi (60), which allow the user to work on a multiple-line "page" of text rather than on a single line.

Interactive editors (61) are emerging into the practitioner community. These are based on "smart" terminals that use an on-board computer which provides high-resolution bit-mapped displays and some sort of fast location selection device such as a "mouse". A variety of options are becoming available, all within a cost range that is reasonable for most software department budgets.

Fairly well understood at this time are more sophisticated editing capabilities that use knowledge of a programming language's syntax to aid in the editing process. These syntax-directed editors (62) do not require sophisticated terminal capabilities but rather they raise the level of man-machine synergism by using knowledge of the language's syntax to guide the editing of a program and assure that only syntactically correct programs result from any editing operation.

Still in the research phase are graphics-based editors. While the hardware aspects of graphics terminals are well known and a wide variety of such terminals are available, how to effectively use the graphics capabilities in support of software development is not fully understood. The TELL system (63) was an early attempt at using graphics in support of software design but the use of graphics in this system was essentially in support of the display of control and data flow charts. These sorts of terminals offer great potential for providing significant support of software development but considerable research is needed.

11. Command Languages

Command language allows the practitioner to identify a tool and activate it against specified data. For the most part, practitioners currently use a traditional operating system job control language for this purpose.

Some systems provide programmable command languages that allow the specification of complex algorithms to control the (conditional and iterative) invocation of tools or the function-like invocation of tools such that the output produced by a tool is used as data in the control program. Examples of such command languages are the sh and csh command languages found in Unix ((64), (65)). More and more, the capabilities of programmable command languages are included in current-day operating systems and software engineering environments.

The increasing availability of sophisticated terminals with bit-mapped displays and a variety of input/output devices raises the question of how to utilize these capabilities in a system's command language. One response has been to define generic command languages that allow the command language developer to define, through tables or other mechanisms, various forms or patterns that should be used in interacting with the user at the command language level. Several possibilities in this direction are discussed in (66).

A command language issue that is currently in the research arena concerns tool aggregation. The basic question is: how can a set of tools be decomposed into reusable tool parts and deployed within a distributed host architecture so that all of the capabilities provided by combinations of the tool parts are effectively and efficiently available to users? Command languages relate to this issue since a command must be compiled into a program that controls the invocation of the tool parts. One solution to this problem has been developed as part of the Toolpack project (25). Alternatives must be investigated, especially for the case in which a particular user of the environment has multiple access sites (office, home, etc.) and needs to obtain the same capabilities from each.

C. CONDUCT OF THIS SURVEY

While the topics chosen for attention here span a broad range of the technologies supporting software development, they do not cover all of the possible topics. We have chosen as broad a spectrum as possible within the limits of our direct knowledge and some important topics (such as management support technology) have been missed.

Examples of technology at the various stages of maturation were found by surveying the current literature. The primary sources were: ACM Computing Surveys, ACM Communication of the ACM, IEEE Computer, IEEE Transactions on Software Engineering, and SIGSOFT Software Engineering Notes.

The assessment given here is corroborated by a study done by the University of Maryland of software engineering technology (67) and a study of testing technology done for the Office of Secretary of Defense (see Volume II of (68)). In these studies, several companies and organizations were surveyed as to the software engineering and testing methods, techniques and tools being used for projects of different types. The results pertain mostly to the state of practice in the surveyed companies and are consistent with the "mature" classification given here.

REFERENCES

1. COMTEC 2000 Study Group, Computer Technology Forecast and Weapon Systems Impact Study (COMTEC-2000), 3 volumes, HQ Air Force Systems Command, Tech Report 78-03, December 1978-July 1979.

2. P. Wegner (editor), Research Directions in Software Technology, MIT Press, 1979.

3. B. Arden (editor), What Can Be Automated?, MIT Press, 1980.

4. S. Redwine, E.Siegel, G. Berglass, Candidate R&D Thrusts for the Software Technology Initiative, OUSDRE (E&PS), May 1981.

5. "Opportunity Assessments," Appendix II in Strategy for a DoD Software Initiative, volume II, 1 October 1982.

6. P. Winetar and R.H. Brown (editors), Artificial Intelligence: an MIT Perspective, 2 volumes, MIT Press, 1979,80.

7. A. Barr and E. Feigenbaum (editors), The Handbook of Artificial Intelligence, 3 volumes, William Kaufmann, 1981-83.

8. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Structured Programming. Academic Press, London, 1972.

9. E. Yourdon and L. L. Constantine. Structured Design. Prentice Hall, Englewood Cliffs, New Jersey, 1979.

10. M. A. Jackson. Principles of Program Design. Academic Press, London, 1975.

11. G. Booch. Software Engineering with Ada. Benjamin/Cummings Pub. Co., Menlo Park, California, 1983.

12. B. H. Liskov and S. N. Zilles. "Specification Techniques for Data Abstractions". IEEE Trans. on Software Engineering, 1:1, March 1975, 7-18.

13. J. R. Cameron. JSP & JSD: The Jackson Approach to Software Development. IEEE Computer Society Press, Silver Spring, Maryland, 1983.

14. M. W. Alford. Distributed Computing Design System Quarterly Review. TRW, Huntsville, Alabama, October 1982.

15. ACM SIGSOFT Software Engineering Symposium on Rapid Prototyping. Software Engineering Notes, 7:5, 1982.

16. See discussion in: W. E. Howden. "Functional Program Testing". IEEE Trans. on Software Engineering, 6:2, March 1980, 162-169.

17. W. E. Howden. "Methodology for the generation of program test data". IEEE Trans. on Computers, C-24, 1975, 554-560.

18. Software Research Associates, Inc., San Francisco, California.

19. Reifer Consultants, Inc., Torrence, California.

20. W. E. Howden. "Functional Program Testing". IEEE Trans. on Software Engineering, 6:2, March 1980, 162-169.

21. S. T. Redwine, Jr. "An Engineering Approach to Software Test Data Design". IEEE Trans. on Software Engineering, 9:2, March 1983, 191-200.

22. L. A. Clarke, J. Hassell and D. J. Richardson. "A Close Look at Domain Testing". IEEE Trans. on Software Engineering, 8:4, July 1982, 380-390.

23. S. J. Zeil. "Testing for perturbation of program statements". IEEE Trans. on Software Engineering, 9:3, May 1983, 335-346.

24. L. D. Fosdick and L. J. Osterweil. "Data flow analysis in software reliability". ACM Computing Surveys, 8:3, September 1976, 305-330.

25. L. J. Osterweil. "Toolpack -- An experimental software development environment research project". Proc. 6th Intern. Conf. on Software Engineering, September 1982, Tokyo, Japan, pp. 166-177.

26. G. Bristow, C. Drey, B. Edwards and W. Riddle. "Anomaly detection in concurrent programs". Proc. 4th Intern. Conf. on Software Engineering, September 1979, Munich, Germany.

27. R. N. Taylor. "A general-purpose algorithm for analyzing con-current programs". Comm. ACM, 26:5, May 1983, 362-376.

28. M. E. Fagan. "Design and code inspections to reduce errors in program development". IBM Systems Journal, 15:3, 1976, 182-211. 29.S. L. Hantler and J. C. King. "An introduction to proving the correctness of programs". ACM Computing Surveys, 8:3, September 1976, 331-353.

30. S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile. "An overview of Affirm -- A specification and verification system". Proc. IFIP Congress 80, October 1980, pp. 343-348.

31. B. A. Silverberg. "An overview of the SRI hierarchical development methodology". In Hunke (ed.), Software Engineering Environments, North-Holland Pub. Co., New York, 1981.

32. See discussion in: R. Balzer, T. E. Cheatham and C. Green. "Software technology in the 1990's: Using a new paradigm". IEEE Computer, 16:11, November 1983, 39-45.

33. See documentation for Unix System III, AT&T Bell Laboratories.

34. J. K. Slape and P. J. L. Wallis. Conversion of Fortran to Ada Using an Intermediate Tree Representation. Tech. Report, School of Mathematics, Univ. of Bath, Bath, United Kingdom.

35. R. A. Freak. A Fortran to Pascal Translator. Report R80-3, Dept. of Information Sciences, The Univ. of Tasmania, Tasmania, Australia, January 1980.

36. J. Martin. Applications Development Without Programmers. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

37. S. H. Caine and E. K. Gordon. PDL -- A tool for software design. Proc. 1975 Natn. Comp. Conf., June 1975, pp. 271-276.

38. R. A. Snowdon and P. Henderson. "The TOPD system for computer-aided software development". In: Bergland and Gordon (eds.), Software Design Strategies, IEEE Computer Society Press, Silver Spring, Maryland, 1979.

39. D. I. Good. "Constructing verified and reliable communications systems". Software Engineering Notes, 2:5, October 1977, 8-13.

40. I. M. Campos. "SARA-aided design of software for concurrent systems". Proc. 1978 Natn. Comp. Conf., June 1978, Anaheim, California, pp. 225-236.

41. J. C. Batz, P. M. Cohen, S. T. Redwine and J. R. Rice. "The application-specific task area". IEEE Computer, 16:11, November 1983, pp. 78-85.

42. B. Boehm. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

43. M. H. Halstead. Elements of Software Science. North-Holland Pub. Co., New York, 1977.

44. Special Issue. Journal of Systems and Software, 2, 1981.

45. P. Freeman and A. I. Wasserman. Comparing software design methods for Ada: A study plan. Ada Joint Programs Office, November 1982.

46. J. R. Dunham and E. Kruesi. "The measurement task area". IEEE Computer, 16:11, November 1983, 47-54.

47. J. Feldman and D. Gries. "Translator writing systems". Comm. ACM, 11:2, November 1968, 77-113.

48. See documentation for Unix System III, AT&T Bell Laboratories.

49. W. Wulf, B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner and B. R. Schatz. "An overview of the production-quality compiler-compiler project". IEEE Computer, 13:8, August 1980, 38-49.

50. M. Ganapathi, C. N. Fischer and J. L. Hennessy. "Retargetable compiler code generation". ACM Computing Surveys, 14:4, December 1982, 573-592.

51. See documentation for Unix System III, AT&T Bell Laboratories.

52. W. Teitleman and L. Masinter. "The Interlisp programming environment". IEEE Computer, 14:4, April 1981, 25-33.

53. A. Goldberg. Smalltalk-80: The Interactive Programming Environment. Addison-Wesley Pub. Co., Reading, Massachusetts, 1983.

54. D. Teichroew and E. A. Hershey. "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems". IEEE Trans. on Software Engineering, 3:1, January 1977, 41-48.

55. M. A. Alford. "Software Requirements Engineering Methodology (SREM) at the age of four". Proc. Compsac 80, October 1980, Chicago, Illinois, pp. 866-874.

56. A. I. Wasserman. "The unified support environment: Tool support for the user software engineering methodology". Proc. IEEE Computer Society SoftFair Conf., July 1983, Crystal City, Virginia, pp. 145-153.

57. W. E. Riddle. "The evolutionary approach to building the Joseph software development environment". Proc. IEEE Computer Society SoftFair Conf., July 1983, Crystal City, Virginia.

58. R. Balzer, T. E. Cheatham and C. Green. "Software technology in the 1990's: Using a new paradigm". IEEE Computer, 16:11, November 1983, 39-45.

59. C. Green, D. Luckham, R. Balzer, T. Cheatham and C. Rich. Report on a knowledge-based software assistant. Report KES.U.83.2, Kestrel Institute, Palo Alto, California, June 1983.

60. See documentation for Unix System III, AT&T Bell Laboratories.

61. N. Meyrowitz and A van Dam. "Interactive editing systems". ACM Computing Surveys, 14:3, September 1982, 321-416.

62. Several articles on syntax-directed editors will appear in: Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Software Engineering Notes, May 1984.

63. P. G. Hebalkar and S. N. Zilles. "TELL: A system for graphically representing software designs". Proc. Compcon Spring 1979 Conf., San Francisco, 1979, pp. 244-249.

64. See documentation for Unix System III, AT&T Bell Laboratories.

65. See documentation for Unix System III, AT&T Bell Laboratories.

66. T. Kaczmarek. "Command language design". In: WIS Implementation Study Report -- Volume III -- Background Information, Inst. for Defense Analysis, Alexandria, Virginia, October 1973.

67. M. V. Zelkowitz, R. Yeh, R. G. Hamlet, J. D. Gannon and V. R. Basili. The software industry: A state of the art survey. Tech. Report, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland.

68. R. A. DeMillo and R. J. Martin. OSD/DDT&E Software Test and Evaluation Project, Phases I and II Final Report. Office of Secretary of Defense, The Pentagon, Washington, D.C.

CHAPTER VI CONCLUSIONS

Software is becoming increasingly important to DoD. It is pervasive in current DoD systems and is becoming more so. Many systems currently in the planning stages cannot operate without software, and long range planning documents indicate that DoD system and mission performance will be increasingly dependent on software. Software reliability and its prediction are not well understood today and their importance will be rising as software grows in volume, criticality, and integrated functionality. The level of software integration is intensifying as systems become more complex and software controls more system functions. In addition to being reliable, software must also be survivable. It must be able to operate perfectly after prolonged periods of dormancy in hostile environments and when portions of the system have been destroyed. Altogether the picture painted by the study of future requirements is one of rapidly rising requirements along a number of dimensions.

Several additional factors may compound this picture of future requirements presented in Chapter II. First, potential adversaries may take unexpected steps to counteract planned systems, so that the actual future requirements may be more complex than those reflected in today's plans. Second, if history is any guide, actually designing and building the systems will turn out to be considerably more difficult than is apparent today. Third, DoD requirements in aggregate will place a more severe demand on the DoD software community than any one such advanced system viewed in isolation might lead one to suppose. Thus, the rapidly rising future DoD software requirements may be even more severe than they appear today.

Conclusion 1: Future DoD software requirements are rising rapidly and becoming increasingly critical to the DoD mission.

A close look at a few major projects generally confirmed numerous prior reports on problems with the current state of practice meeting current requirements. Problems with budget, schedule, requirements, staffing, and product quality were found. Nevertheless, in these few projects capabilities were being fielded and, while it was often a struggle, current requirements generally are eventually being met.

Conclusion 2: The current software state of practice is having difficulties meeting current DoD requirements.

The study of fourteen software technologies indicates that bringing a technology to the point of maturation where it is popularized and disseminated to a large portion of the technical community generally has taken 15-20 years. Other studies indicate that 4-8 additional years may be required to propagate that technology throughout a large organization. Technological concepts and ideas that do not need computerized support or a change in the using community's mind set can mature faster. Particularly important to technology transition are a recognized need, a receptive target community, and a believable demonstration of cost/benefit. Well-designed channeling of attention and support, an articulate advocate, prior success, incentives, technically astute managers, readily available help, latent demand, simplicity, and incremental extensions to current technology were also identified as facilitators. Most significant among the technology transition inhibitors are the time it takes to transfer a technology internally, high cost, contracting disincentives, psychological hurdles, and the desire by programmers to "fiddle" with a technology that is too easily modified. Technology originators most often inhibited transition by small, simple mistakes that were easy to correct once identified. The Government has also not always facilitated technology transition.

Conclusion 3: Software technologies have taken significant time to reach widespread use -- 15 to 20 years.

Conclusion 4: The possibility exists to facilitate and accelerate software technology transition.

The state of the art in software technology includes many new or immature technologies that could improve the state of practice if brought to maturity, made usable, and used. In Chapter V several examples of these were reviewed.

Conclusion 5: Many immature or unused software technologies exist that offer potential opportunities to improve the future state of practice.

Conclusions 1, 2, and 3 point to an unfortunate combination of conditions:

o Future DoD software requirements will rapidly increase not just in terms of size, functionality and pervasivess, but also in terms of reliability, complexity, and criticality to DoD mission.

- The current software state of practice has frequent problems meeting today's requirements including budget, schedule, product quality, and requirements problems.

- Innovations in software technology have been taking a long time to mature and become widely used --typically 15 to 20 years.

These combined with consideration of two additional points lead to the conclusion that a potentially serious software problem exists. First, the state of practice relevant to meeting a requirement in a given year is the state over the several preceding years while the system was being specified, designed, and built. Second, requirements in the aggregate will present a significantly greater challenge than individually. As embedded software proliferates in non-DoD industries such as aviation, medicine, and communication, DoD could be faced with intense competition in the commercial marketplace for skilled human resources. Thus substantial future gap between software-related DoD requirements and the software state of practice is a real threat particularly given the criticality of software in many planned DoD systems.

Conclusion 6: A real potential exists for a critical future gap between DoD system and mission requirements and the future software state of practice's ability to meet them.

Conclusions 4 and 5, however, point to some opportunities that if properly exploited by DoD might help close this gap. Many immature technologies exist and their maturity and use could be accelerated. Problems in the current state of practice, however, indicate that technology is not the only issue; management, acquisition, and personnel are also areas of concern.

Conclusion 7: Opportunities exist for DoD to help close the potential future software gap between requirements and the ability to meet them by accelerating technology transition combined with concern for management, acquisition, and human resources.

APPENDIX A

METHODOLOGY

This section describes the approach and methods used to estimate future software requirements of DoD weapons systems. Two methodolgies were employed: (1) a compilation of current and planned major DoD programs and (2) a preliminary analysis of DoD long range planning documents. Certain factors limited data collection and analysis.

1. Survey

Data regarding future DoD software requirements were obtained through a survey of select key DoD programs, under the assumption that an understanding of present requirements might lead to future requirements projections.

The Report of the Secretary of Defense Casper W. Weinberger to the Congress on the FY 1985 Budget, FY 1986 Authorization Request and FY 1985-1989 Defense Programs, published February 1, 1984, identifies 160 key programs. This report contains brief descriptions of major DoD programs and, in most cases, development and procurement funding for fiscal year 1983-86.

Of the programs identified in the report, at least 120 (or 75%) were determined to have a significant software component. The program element number, in most instances, was used to obtain program descriptions from the Defense Technical Information Center's (DTIC) Work Unit Information Summary (WUIS) file. When program element numbers were not easily identified, program titles were used to search the DTIC WUIS. Program descriptions were used to identify those programs with a software development component. In some instances, there was difficulty in obtaining WUIS project summaries, therefore other sources such as DMS Market Intelligence Reports (1) and telephone calls to project offices were relied on.

In addition, other programs with software development components were identified in the online DMS Market Intelligence Reports files where the "Computer" was listed as a major type of equipment.

A cursory examination of 52 "new" programs with software development requirements scheduled to begin funding in 1985 or later was conducted. These 52 programs were selected from a total of 65 programs listed in the RDT&E Annex of the Five Year Defense Program (dated 1 February 1984) and scheduled to begin funding in 1985 or later.

Thus, a total of 185 programs, including the 120 programs identified from Secretary Weinberger's report, the 13 from the search of the data base, and 52 from the RDT&E Annex, were given preliminary reviews.*

For each of these 185 programs, information gathered included: Program Title, Program Element Number, DoD Office, Short Description of Software Activity, and Development Funds for the years 1984 through 1989 (usually RDT&E funds).

The 185 programs were categorized by type of program to facilitate comparison with future technologies identified by the long range plans. Appendix B lists the categories and the relevant programs under each.

A comparison of the total development funds (usually RDT&E) for 1984, 1985, and 1986 for these 185 programs with the total RDT&E budget for those years revealed that 45% of the total

*The Air Force has supplied a list of 238 Air Force programs with software requirements.

RDT&E program, in terms of money, had been captured by the list of 185 programs. The assumption was that major software developments and modifications would appear in RDT&E funds. Software maintenance included in operation and maintenance (O&M) funds were not included in the list. Although the list was not comprehensive, it did include a broad cross section of programs, as evidenced by the categories in Appendix B. Since most of the programs had been described in the Report of the Secretary of Defense Casper W. Weinberger to the Congress on the FY 1985 Budget, FY 1986 Authorization Request and FY 1985-89 Defense Programs, it was also assumed that the major programs (in terms of money and importance to the national defense) had been included.

A telephone survey of a random sample of these programs was attempted but proved unsuccessful. In depth information about software size, cost, and functionality was difficult to obtain in a telephone survey of DoD Programs Managers in the limited time available. Software cost data is not readily available in most cases. David Fisher also reported difficulty in isolating software costs in his 1974 study Automatic Data Processing Costs in the Defense Department, Institute for Defense Analyses. Some difficulty was experienced in obtaining complete information because some of the programs surveyed were in the procurement process, others had security restrictions, or the program contract was an A-109 (2) competitive procurement.

2. Examination of DoD Long Range Plans

A review of selected DoD long range plans was conducted to determine the scope of software requirements. The DoD does not always make available plans of programs more than five years in advance, but the DoD Service long range plans describe technologies, functions and systems that are projected for the future. An analysis of these requirements for a software component allowed for a first attempt at examining future needs.

Appendix C lists the long range plans which were examined. An attempt was made to include a balanced representation of military service plans for maintaining the United States' military posture until the year 2000.

Plans were identified through a number of sources. The DoD's Optimis (Operations Management Information System) database was searched for references to Service long range plans and long range planning offices. It was found that the Army has an office dedicated to developing an exchange of ideas on long-range planning among the Services. This office produced a list of Service offices concerned with long range planning. Appendix D lists the offices and people contacted for each Service. Other documents were identified through an online search of the Defense Technical Information Center's Technical Report file.

Each Service plan was examined for future technologies which were listed separately and analyzed for software components. The number of technologies requiring software was compared to the total number of technologies and percentages calculated.

Plans were also examined for statements specifying future reliability requirements for software and for examples of new software applications which are mission critical.

One multi-service space-related plan, the Battle Management, Communication, and Date Processing, volume of the Defense Technical Study Team (DTST) study was examined as an example of a future planned DoD program involving massive amounts of software (the Strategic Defense Initiative).

REFERENCE

(1) DMS, Inc., 1700 N. Moore Street., Suite 1230, Arlington, VA, 22209.

(2) Circular A-109 Major Systems Acquisition, Office of Management and Budget, April 1976, UNCLASSIFIED.

APPENDIX B

PROGRAMS CATEGORIZED*

The first part of this list follows the categorization in the Report of the Secretary of Defense Casper W. Weinberger to the Congress on the FY 1985 Budget, FY 1986 Authorization Request and FY 1985-89 Defense Programs and comprises mainly weapons necessary for Land, Naval, Air, and Nuclear combat. The remainder of the programs have been organized into the non force-specific categories of Communications; Command and Control; Intelligence and Electronic Warfare; Combat Support, Combat Service Support; and Research and Development. Programs are listed only once even though they may qualify for more than one category (Command and Control, and Communications, for example). The categories are as follows:

Land Forces

- Close Combat
- Land Force Aviation
- Air Defense
- Artillery Fire Support
- Deep Interdiction

Naval Forces

- Anti-Air Warfare
- Anti-Submarine Warfare

Tactical Air Forces

Nuclear Forces

- Strategic Offensive Forces
- Strategic Defensive Forces

Communications

Command and Control

Intelligence and Electronic Warfare

Combat Support

Combat Service Support

Research and Development

*Note: The 52 programs from the FYDP (Five Year Defense Program) RDT&E Annex are not included here because the FYDP is classified. Since they all begin in 1985 or later, they would be included in the Research and Development category, if they were here.

LAND FORCES

Close Combat

Bradley Fighting Vehicle (BFV) System
Light Armored Vehicle
M-1 Abrams Tank
TOW Missile System

Aviation

AH-64 (Apache)
Hellfire
Joint Services Advanced Vertical Lift Aircraft (JVX)
Light Helicopter Family (LHX)

Air Defense

Chaparral
Improved Hawk (IHAWK)
Patriot
Product Improvement Vulcan Air Defense System
Short Range Air Defense Command and Control (SHORAD C^2)
Stinger

Artillery Fire Support

Advanced Field Artillery Tactical Data System (AFATDS)
Artillery Computer System
Modular Universal Laser Equipment (MULE)
Multiple Launch Rocket System (MLRS)
Multiple Launch Rocket System (MLRS) Terminally Guided Submunition (TGSM)
Remotely Piloted Vehicles
TACFIRE

Deep Interdiction

Joint STARS (Surveillance and Target Attack Radar System)
Joint Tactical Fusion System (JTF)
Joint Tactical Missile System (JTACMS)

NAVAL FORCES

Anti-Air Warfare

Aegis Missile System
DDG-51 Guided Missile Destroyers
Joint Tactical Missile System
Rolling Airframe Missile System

Anti-Submarine Warfare

Anti-Submarine Warfare Program
Light Airborne Multipurpose System (LAMPS)
MK-48 Torpedoes

MK-50 Torpedoes
Rapidly Deployable Surveillance System (RDSS)
SH-60B Lamps MK II
Tactical Towed Array Sonar (TACTAS)
Vertical Launch Anti-Submarine Rocket

TACTICAL AIR FORCES

Advanced Medium Range Air-to-Air Missile (AMRAAM)
A-6E Intruder
A-6E Intruder Improvements
AV-8B (Harrier)
F-14 (Tomcat)
F-15 (Eagle)
F-16 (Fighter Falcon)
FA-18 (Hornet)
HH-60 D/E Nighthawk Modernization
Laser Maverick
Low Altitude Navigation and Targeting Infrared (LANTIRN)

NUCLEAR FORCES

Strategic Offensive Forces

Air Launched Cruise Missile
B-1 Bomber
B-52 Bomber
Follow-on Basing Technology
ICBM Modernization
Minuteman Modernization
Peacekeeper Missile and Basing
Small ICBM and Mobile Launcher
Trident II Missile

Strategic Defensive Forces

Air Defense
Space Defense
Strategic Defense Initiative

COMMUNICATIONS

Air Force Satellite Communications
Army Data Distribution System (ADDS)
Autovon/Autodin
Ballistic Missile Early Warning System (BMEWS)
Defense Communications System
Defense Data Network
Defense Meteorological Satellite Program (DMSP)
Defense Satellite Communications System (DSCM)
Defense Switched Network (DSN)
European Command, Control, and Commuications System (EUCOM C^3)
Ground Mobile Forces Satellite Communications
Ground Wave Emergency Network

High Technology Light Division
Joint Tactical Communications (TRI-TAC) Program
Military Satellite Communications
Military Strategic and Tactical Relay (MILSTAR)
Minimum Essential Emergency Communications Network (MEECN)
NAVSTAR Global Positioning System (GPS)
Navy Fleet Satellite Communications System
Norad Improvements
Nuclear Detection System (NDS)
Pave Paws
SAC Digital Network
Satellite Control Facility
Satellite Early Warning System
Secure Voice Improvement Program
Single Channel Ground and Airborne System VHF (SINCGARS V)
TACAMO E-6A Aircraft

COMMAND AND CONTROL

Advanced Airborne Command Post (AACP)
Advanced Digital Optical Control System (ADOCS)
Air Launched Control System
Conus Over the Horizon Radar System (OTH)
E-3A Airborne Warning and Control System (AWACS)
E-2C Hawkeye
European Command, Control, and Communications System (EUCOM C^3)
Integrated Tactical Surveillance System (ITSS)
Joint Deployment System
Joint Interoperability of Tactical Command and Control Systems (JINTACCS)
Maneuver Control System (MCS)
National Military Command System (NMCS)
Naval Command and Control System
Tactical Air Control System Improvement
World Wide Military Command and Control System (WWMCCS)
WWMCCS Information System (WIS)

INTELLIGENCE AND ELECTRONIC WARFARE

Airborne Self-Protection Jammer (ASPJ)
EA-6B Prowler
Electronic Warfare Technology Program
Joint Surveillance System
Joint Tactical Information Distribution System (JTIDS)
Missile Warning and Attack Assessment Sensors
Naval Tactical Data System (NTDS)
Operational Tactical Data System
Pave Tiger
Precision Location Strike System (PLSS)
Space Defense
Space Track
Tactical Information Processing and Interpretation (TIPI)-Marine Air/Ground Intelligence System (MAGIC)

COMBAT SUPPORT

Chemical Warfare Defense Program
Combat Support Equipment
Next Generation Weather Program
Wide Area Anti-Armor Munitions

COMBAT SERVICE SUPPORT

B-1 Weapon System Trainer
C-17 Cargo Airlift
Chemical/Biological Detection Warning/Sample Material Concept
Education and Training System
MK 92 Fire Control Maintenance Trainer
Modular Automatic Test Equipment (MATE)
Non System Training Devices

RESEARCH AND DEVELOPMENT

Advanced Materials Program
Consolidated Space Operations Center (CSOC)
Directed Energy Technology Program
Manufacturing Technology Programs
Medical and Life Sciences Program
Nuclear Weapons Effects Research Program (NWE)
Space Surveillance Technology
Strategic Computing Program
Survivable Radar Station/Sites
Very High Speed Integrated Circuit (VHSIC)

APPENDIX C
BIBLIOGRAPHY OF LONG RANGE PLANS

ARMY

Airland Battle 2000, HQ US Army Training and Doctrine Command, Ft. Monroe, VA, August 1982. UNCLASSIFIED.

A Concept of a Future Force, Charles W. Taylor, Army War College Strategic Studies Institute, November 1981. UNCLASSIFIED.

Long Range RDA Plan FY 83-97, Office of the Deputy Chief of Staff for Research, Development and Acquisition, U.S. Army, July 1981, SECRET.

Prototype Army Long-Range Appraisal (PALRA) Enclosure I--Long-Range Requirements, BDM Corp, December 1981. SECRET.

Technology Opportunities for Focus 21 (Appendix to Focus 21 document in progress by Army and Air Force) no date. SECRET.

AIR FORCE

AFSC Vanguard Planning Summary, DCS/Plans and Programs, HQ Air Force Systems Command, Andrews Air Force Base, Washington, DC, December 1983. SECRET.

Air Force 2000: Air Power Entering the 21s Century, US Air Force Office of the Chief of Staff, June 1982. SECRET.

SHORAD Systems For The Year 2000, Office of the Assistant Chief of staff Studies and Analysis, U.S. Air Force, October 1983. SECRET.

NAVY

Future Battle Forces: Initiative and Opportunites, Delex Systems Inc., February 1983. SECRET.

Marine Corps Long Range Plan (MLRP), MARCORPS-2000, Deputy Chief of Staff for Plans, Polices, and Opera..ons, Department of the Navy, HQ U.S. Marine Corps, May 1982. SECRET.

Master Plan for Embedded Computer Resources, Headquarters Naval Material Command, April 1982. UNCLASSIFIED.

Navy Command and Control Plan, Office of the Chief of Naval Operations, Washington, DC, March 1983. SECRET.

Post 1985 Naval Command and Control and Communications Requirements Study, Naval Underwater Systems Center, May 1978. SECRET.

Project 2000, Office of the Chief of Naval Operations, June 1974. SECRET.

Sea Plan 2000, Department of the Navy, April 1979. SECRET.

Surface Ship Combat System Master Plan, Chief of Naval Operations, OP-03, March 1984. SECRET.

DOD

Battle Management, Communications, and Data Processing, volume v of the Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles, Defensive Technologies Study Team, October 1983. UNCLASSIFIED.

Military Space Systems Technology Model, Volume I. Appendix. Long Range Planning Objectives (Non-Validated), General Research Corp., January 1982. SECRET-RD

NATO

Proceedings of the 23rd DRG Seminar, Operational Research for the Selection and Design of Future Military Systems, North Atlantic Treaty Organization, September 1982. CONFIDENTIAL.

APPENDIX D

DoD Long Range Planning Offices Contacted

<u>ARMY</u>

Deputy Chief of Staff for Operations and Plans
Strategic Plans and Policy
Long Range Planning
3B521 Pentagon
Washington, D.C. 20310

Major Edward Lauer
694-8241

Assistant Chief of Staff for Information Management
Plans and Programs Division
1D679 Pentagon
Washington, D.C. 20310
(formerly Deputy Chief of Staff for Plans and Operations,
C-4 Plans and Programs-prior to 5-11-84)

Charles Colello
697-0534

Deputy Chief of Staff for Operations and Plans
Force Development Directorate
Doctrine, Force Design and Systems Integration Division
2B546 Pentagon
Washington, D.C. 20310

*Major Mike Kendall
695-1861

Deputy Chief of Staff for Research, Development, and
Acquisitions
Army Research Technology
3E429 Pentagon
Washington, D.C. 20310

*Lt. Col. William H. Freestone
697-0296

*-Briefing

AIR FORCE

Deputy Chief of Staff for Plans and Operations
Directorate of Plans
Deputy Director for Planning Integration
Long Range Planning Division
5D230 Pentagon
Washington, D.C. 20330

Lt. Col. William R. Caldwell
697-3717

Air Force Systems Command
Project Vanguard
Andrews AFB, MD

AV 858-3307

NAVY

CNO Executive Panel
Long Range Planning
c/o Center for Naval Analyses
2000 N. Beauregard St.
Alexandria, VA 22311

Commander Robert Harris
694-8422

Command and Control Directorate
Information Systems Division
5E571 Pentagon
Washington, D.C. 20350

Capt. Andy Tate
695-6792

JCS(JOINT CHIEFS OF STAFF)

Joint Staff J-5
Strategy Division
1E965 Pentagon
Washington, D.C. 20310

Col. J.S.V. Edgar
695-5630

APPENDIX E

Analysis of Airland Battle 2000 for Future Software Requirements

Functional Area	Capabilities Counted	Capabilities with Software Components	% of Capabilities with Software Components
Command and Control	17	15	88%
Close Combat	18	14	78%
Fire Support	21	13	62%
Concept for Air Defense	18	16	89%
Intelligence and Electronic Warfare	18	15	83%
Communications	8	8	100%
Combat Support, Engineer, and Mine Warfare	31	15	48%
Combat Service Support	13	5	38%
Army Aviation	28	23	82%

FUTURE SOFTWARE DEVELOPMENT NEEDS

Command and Control

- survivability - chemical, biological, and radiological protection and self-decontamination
- communications - smaller, lighter; EMP and ECM-resistant; reduced probatility of intercept; cryptographically secure; reliable; transparent to user
- management system - common DBMS for passing information between systems
- data distribution - integrated microprocessors combined with communications allow filtering and multiechelon distribution of data to provide information appropriate to each level of command
- graphics - electronically produced, visually-displayed, rapidly and securely transmitted to disseminate operations plans and orders
- command posts - smaller in terms of size, not function operate continuously in all combat environments (nuclear, biological, chemical, and electronic lethality dictate smaller)
- automation (attributes)
  - local and rapid programming capability
  - voice recognition
  - multiple path message routing
  - access to all supporting databases
  - common decision graphics
  - fail-soft degradation
  - usable for peacetime administration and training
  - hard copy point-to-point messages
  - word processing
  - support decision-making only

Close Combat

- o Fire-and-forget precision-guided missiles
- o Electronic deception systems
- o Survivable C^2 systems
- o Obstacle neutralization systems
- o Robotics
- o Directed-energy weapons and support systems
- o Lethal and nonlethal chemical weapon systems
- o Special effects weapons
- o Multicapable weapons
- o Continuous intelligence preparation of battlefield proving a comprehensive and complete database of enemy, terrain, and obstacle information
- o Simplistic man-machine interface
- o Simplistic operation of complex systems
- o Training to fight on the integrated battlefield

Fire Support

- o Field Artillery Attack Systems to perform the fire support control and coordination function (FSC^2)
- o Explore non-electronic means of communications
- o Electronic deception systems
- o Automated Assistance
- o Support C^2 of autonomous weapons systems
- o Identify and establish human interface nodes
- o Establish Force level DBMS
- o Weapons platforms capable of autonomous operations
- o Capability to acquire long range passive targets

- o Explore robotics and materiel handling equipment (MHE)
- o User selected multi-function black boxes
- o Remote resupply/recovery
- o Passive sensors-combat vehicle mounted
  - Nettable to FSE
  - Capable of autonomous support to fire units

<u>Concept for Air Defense</u>

- o Rail Gun (Hypervelocity)
- o Smart or maneuvering projectiles
- o Laser
- o Particle beam
- o Microwave
- o Non nuclear EMP
- o Jammers
- o Shoot on the move
- o Fire and forget
- o Self-initiating
- o Antiair mines and barriers
- o IR
- o Acoustics
- o UV
- o RF
- o Visible

Intelligence and Electronic Warfare (IEW)

- o Rapid access to intelligence capabilities (national and multi-service command levels) both during peace and war
- o Systems with no unique physical or electronic signatures to distinguish from non-IEW systems
- o Unique IEW skills reduced to lowest level possible through use of robotics and AI systems
- o Require less technical and training intensive skills from human operators
- o Distributed intelligence data base
- o Reliable, redundant communications system
- o Less vulnerable sensor systems
- o Automated threat projections
- o Continual, real time access to national systems
- o Specific-mission jammers
- o Electromagnetic signature simulators
- o Voice recognition devices
- o Automatic language translators
- o Digital terrain data systems
- o Robotics and AI

Communications

- o Simple to operate, self-diagnostic, self-repair
- o No unique signature
- o Power systems that are mobile, reliable, have little or no acoustic, electronic or thermal signature and have reduced dependency on fossil fuels.
- o Accurate position and location devices that provide information automatically to C^2 databases

- Automation of network management and switching equipment that will provide automatic circuit restoration

- Reliable and accurate data communications systems that will allow distributed data processing and multiechelon data distribution

- Alternative transmission means (laser, infrared light) counter effects of directed energy

- Development of robotics and AI systems to facilitate communications.

<u>Combat Support, Engineer and Mine Warfare</u>

- Light-weight, low-cost mine detection and neutralization system permitting every assault vehicle to detect, and report mines without operator assistance

- Standoff detection and neutralization systems

- Non-explosive neutralization systems for reduction of conventional and remote mines located in areas where explosive destruction is not practical or where clearing of bypassed or breached fields must be done.

- Simple to erect, low cost fixed bridges-remotely delivered

- Remotely deployed intelligent, robotic, automatic weapon stations

- Local and wide area weather control

- Topographic systems-position reporting and recording and terrain information and video terrain displays. Automated terrain database

- Real time remote terrain sensors

- Automated terrain and weather analysis with graphic output

- Beamed power to point of use from collection stations

- Computer monitoring of structures for wear and damage

- o Automated photogrammetric input and computer-based civil engineering techniques for road design in difficult terrains
- o Engineer vehicles with same base vehicle as the close combat force, externally mounted, externally accessible tool boxes. Integrated personnel decontamination chamber and internally operable cybernetic tools.
- o Robotics and sensors to maintain operations during rest periods
- o Fuses which identify friend or foe, can be command, time, or time extended detonated and produced at low cost
- o One-shot cratering charge capable of hand emplacement or remote delivery.

Combat Service Support (CSS)

- o Robotics
- o Secure communications
- o Real time intelligence
- o Remote processing
- o On board test equipment

Army Aviation

- o Self-deploy
- o Continuous operation is all types of terrain, weather, battlefield environments
- o Sufficient C^2 and CSS to perform or as part of combined arms force
- o Systems have no unique signature
- o Refinement of man-machine interface
- o Robotics, microelectronics, miniaturization
- o Electronic mission aerial platforms and aeroscouts

- Electronics deception systems
- Real time C^2 intelligence databases
- Aerial retransmission and data burst systems
- Field artillery aerial observation platforms and aeroscouts
- Area fire weapon system
- Sophisticated target acquisition systems
- Rapidly transport weapon systems, supplies and personnel around battlefield
- Air to Air Weapons
- Visual identification
- Acquire and destroy enemy air defense weapons
- Aerial platforms electronically detect movement of enemy aerial forces
- Terrain, weather sensor packages carried by aerial platforms
- Dedicated aerial target acquisition systems
- Aerial transmission, reception, and retransmission communications systems
- Self contained, automated remote communication units airlifted
- Delivery means for propaganda

APPENDIX F

NAVY C^2 PROGRAMMATIC ACTIONS

<u>New Actions Requiring Software</u>

Two-site ELF (Extremely Low Frequency) communication system

Surviving airfield location system

Reconstitutable communications

"Seize-key" interface for TACAMO uplink

TW/AA (Threat Warning/Attack Assessment) to TACAMO alert and standby airfields

ECC (Enduring Command Center) development for post-attack C^2

ELF receivers to SSNs (Attack Submarine, Nuclear Powered)

Alternative two-way communications for SSN (DS) (Direct Support)

SSBN capability to search rapidly HF band

Afloat commanders timely access to relevant imagery materials

Programs/upgrades for Arctic communications

Interim command displays

NCCS (Navy Command and Control Systems) Ashore Software Support Facility

Common modular software

TFCC/NIPs (Tactical Flag Command Center/Naval Intelligence Processing Center) interface

C^2 Processor

TADIXS (Tactical Data Information Exchange System)

Afloat Correlation System Program

Link 11 operational improvements

Link 11 improvements compatible w/communication/crypto equipment

Link 11 improvement interoperable w/navy/joint/allied links

Automatic navy input systems development

Automatic gridlock capability

JNIDS (Joint National Intelligence Dissemination System) testbed

New protocols for testing of national/theater sensor systems

EWC (Electronic Warfare Coordinator) module program

Afloat ESM (Electronic Warfare Support Measures) and combat systems

Generic ECM Electronic Countermeasures) Systems

Modified Actions Requiring Software

EC-130 navigation/communication system

Desk top computer based TSS (Tactical Support System)

Essential Existing Program Actions Requiring Software

Joint EHF SATCOM (MILSTAR) terminals

COMSEC (Communications Security) programs

JTIDS (Joint Tactical Information Distribution System) program

Multiple channel HF AJ

Implement CORS (Composite Operational Reporting System)

OTCIXS (Officer in Tactical Command Information Exchange System)

NAVSTAR GPS (Navigation Satellite Time and Sharing Global Positioning System)

TFCC (Tactical Flag Command Center)

SHF SATCOM terminals

NCSS Ashore Upgrades

OSIS (Ocean Surveillance Information System) improvements

Probe Alert development

LEASAT (Leased Satellite)

NTDS (Naval Tactical Data System) Upgrades

EMI (Electromagnetic Interference) control efforts

APPENDIX G

Technology Case Studies

(1) John Bailey. Cost Model Technology Transition. May 1984.

(2) Paul C. Clements, et al. Case Studies of Software Engineering Technology Transfer. Tech. Memorandum, Naval Research Laboratory, April 1984.

(3) Richard A. DeMillo. Compiler Technology Insertion Network Study. May 1984.

(4) John H. Manley. Technology Case Study: Software Engineering Concepts. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, May 1984.

(5) John H. Manley. Technology Case Study: Software Metrics. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, April 1984.

(6) John H. Manley. AFR 800-14 History. Tech. Memo, Computing Technology Transition, Inc., Madison, Connecticut, May 1984.

(7) Ann Marmor-Squires. Formal Software Verification as an Example of Software Technology Transfer. May 1984.

(8) Ronnie J. Martin. DOD-STD-SDS: The Development of a Standard. May 1984.

(9) Samuel T. Redwine, Jr. Structured Programming: A Technology Insertion Case Study. Computer and Software Engineering Division, Institute for Defense Analyses, May 1984.

(10) William E. Riddle. "The Magic Number Eighteen Plus or Minus Three: A Study of Software Technology Maturation." ACM SIGSOFT Software Engineering Notes, 9, 2 (April 1984). (Includes case studies of Unix, Smalltalk-80, and SREM.)

(11) William E. Riddle. Knowledge-based Systems as a Case Study in Software Technology Maturation. SDAM/15, software design & analysis, inc., April 1984.

(12) William E. Riddle. Abstract Data Types as a Case Study in Software Technology Maturation. SDAM/16, software design & analysis, inc., April 1984.

(13) David Weiss. Time Line for Development and Transfer of SCR Methodology. February 1984.

Cost Model Technology Transition

John Bailey
Software Metrics, Inc.
Falls Church, Virginia

May 1984

ABSTRACT

Three well-known cost models are described historically and technically. Also, a brief account of the history of software estimation at IBM FSD is included. Although many organizations are reluctant to describe in detail the techniques they use for resource estimation, since privacy of this information is essential for successful competition, some useful observations by both modelers and model users are included here.

Organization of Report

The histories of two black-box models, PRICE S (1) and SLIM (2), and an open model, COCOMO (3), are the topics of much of this report. These are described in separate sections with cross-referencing between the sections for comparison where appropriate. Following these are some observations by Claude Walston of the advent of software estimation during his tenure at IBM Federal Systems Division's Software Engineering Cost Unit. Following this account are some overall conclusions about the nature of inserting the technology of software cost modeling into the industry. A final section summarizes some of the pros and cons facing the further development, advancement, and use of software cost models.

PRICE S History

The history of PRICE S dates back to 1961 when a parametric hardware cost model, PRICE H, was first developed at RCA for cross-checking estimates of hardware development cost. This model was used manually through the 60's on internal projects at the Morristown Missle and Radar Division. During this time it also migrated to several other groups within RCA, such as Government Communication Systems, Astro-Electronics, and the Automated Systems Group.

Around 1969, PRICE H was automated to simplify its use, however it was still used mainly as a cross-check for more conventional techniques. In 1971, several government customers became interested in the utility of this computerized programmed estimation model and contracted with RCA for the use of the model to estimate proposed hardware development work. Among these customers were the Air Force, the Navy, and NASA. Since RCA helped in this way to establish the expected cost of a project, they were not able to bid on these projects. To eliminate the disparity, in 1975, RCA formed PRICE Systems as a separate business entity to market models and to make them available to competitors.

In 1977 Dr. Robert Park implemented a parametric software estimation model using the methods employed in PRICE H as a point of departure. When PRICE Systems announced the software model, there were twelve immediate customers for it. These were all customers of the PRICE H model who needed the additional capability of having a software estimation tool. This appears to be a unique example of a pre-captured market (see Editorial, below).

Further refinements to the PRICE S model occurred in 1980, with the introduction of the life cycle model, PRICE SL, in 1981 with a second revision to the basic model, PRICE S3, and in late 1982, with a revision to the life cycle model, PRICE SL2. Currently, a micro-computer version is being developed which will provide new output display capabilities, including color and graphics.

At this time there are about thirty-five customers, with more industrial customers than government ones. A customer pays $38,000 per year or $3,850 per month for the license plus dial-up charges. Use of the software on one's own PRIME computer costs $58,000 per year. Second source access to the model is available from some time sharing companies. Training from RCA is additional. Included in the current list of customers are NASA, FAA, Air Force, Army, Navy, Boeing, Ford Aerospace, General Dynamics, General Electric, Grumman, IBM, ITT, Litton, Lockheed, Martin Marietta, McDonald Douglas, Perkin-Elmer, Sperry, Texas Instruments, TRW, and Westinghouse. It is difficult to assess the extent of use the model receives at each customer's site, since customers are only identified at the company level (no additional user site fees are required within a customer company).

An interesting outgrowth of the PRICE user's group and others was the founding of the International Society of Parametric Analysts in the late 70's. This society recognizes estimating as a profession and collects and shares tools and methods among its several hundred members. It has also developed a language to share approaches to estimation problems without revealing corporate secrets.

Technical Summary

The major input to the model is the size of the final executable product measured in machine executable instructions. This has apparently remained unchanged from the first versions. PRICE Systems provides example expansion ratios to accommodate estimates in high level source lines. According to Dr. Park, however, the specific language used is not overly emphasized in the model since the coding effort is typically quite small when compared with all other activities required for software development. In addition to a size input, two empirical indices are computed from the user's data base of project attributes and historical data. These represent a resource factor and a complexity factor.

According to Dr. Park, PRICE S attempts to simulate experienced estimators and managers through the composition of several sub-models. Through interviews and experimentation a simulation/emulation data base was developed which is used by the model. In this way, it seems parallel to COCOMO, and this is partially confirmed by Park. However, he maintains that a wider variation in productivity is often obtained when adjusting the input parameters to PRICE S as compared with COCOMO.

Editorial

The immediate and automatic customer base for PRICE S mentioned earlier is the only example of sudden external acceptance for a new technique which was discovered in the research for this report. It is assumed by PRICE Systems that these customers had no other analytic methods of estimating software at that time. However, there are at least two other important reasons which could have forced this model into rapid use. First, the government was known to have used PRICE H to cost several of their hardware RFP's. Now that the software model was available, it would probably also be used by the government for the same purpose. This would seem reason enough to have access to the same techniques for the purpose of writing proposals to the government for software development. Second, there was probably concern on the part of each PRICE S customer that it not lack any of the capabilities of its competitors if at all possible, implying that several companies would invest in this tool even before it had established any kind of track record.

Since PRICE Systems is a business entity, its product is proprietary and costly. Unfortunately, this restricts experience with the model to serious users who are probably not very likely to share the specifics of their results. This observation is also applicable to the SLIM model, which is also the basis for a business enterprise. Nevertheless, although the marketing department at PRICE Systems admits that the black box nature of the product discourages some potential clients, it maintains that it also attracts some customers. One would probably expect that the ability of a tool to reduce the complexity of a job, by reducing the amount of detail which must be handled and analyzed, is highly desirable. For some, however, this is apparently even true when much of the control over the behavior and use of these details is taken away.

Dr. Park states that significant revisions, even though they may represent substantial improvements, are difficult to promote since many users are now comfortable with the current model and might experience unwanted side-effects when using a new version. This is probably a result of hiding the mechanics of the model from the users, but on the other hand, users who are interested in controlling and developing their own models of the cause and effect relationships at their sites will probably not rely on PRICE in the first place. One of the advantages of producing the micro-computer version of the model is that it will allow many other changes designed to attract new users without necessarily being compatible with the PRIME-based version, according to Park.

One important aspect of any model or tool is whether it is general enough to continue to be useful and relevant as techniques, standards, and practices evolve. According to Dr. Park, it is unknown at this time whether PRICE S has been used successfully with software development approaches other than a traditional waterfall cycle, such as rapid prototyping or software assembly from components. However, he points out that iterative enhancement has been projected successfully with the model.

CCCOMO

History

Early development of the COCOMO model was begun by Dr. Barry Boehm in 1976 at TRW DSG, where he is currently Chief Engineer of Software and Information Systems Division. At that time, TRW was using the Wolverton model (4) to some extent and had some experience with the early SDC data (5) and the Aron model (6). This date coincides with the appearance of Putnam's work at COMPCON that Fall (7).

Initially, relationships (functional forms) between software cost drivers and actual data were sought for ten internal and ten external software development projects. In

1977, TRW increased its support for this work, due in part to initial demonstrations of its utility and also because several other resource modeling efforts were being published (Doty (8), Walston and Felix (9), Boeing (10), and Putnam (11)).

In that year, the initial COCOMO model was developed which was similar to several of the above models in different ways. For example, it used an adjusted base line like the Doty model, but was also phase sensitive like the Boeing model. In that year and the next, COCOMO was required for use by the immediate 800-person organization. The estimates were not binding on the managers, but served as a cross check. These estimates were accomplished by hand by Boehm during this period, and the results are described as credible.

Later in 1978 the model was computerized due to the increased demand which resulted from its initial success. After that, COCOMO was required as part of the proposal for all software projects in the division which were expected to take more than five man-years. In addition, managers were to use at least one other technique.

Technical Summary

COCOMO is a composite model which means that it incorporates a combination of functions (linear, multiplicative, analytic, and tabular) to estimate software effort from project attributes. Althouth both SLIM and PRICE S are proprietary models, it would appear that they, too, are composite models. COCOMO is an open model and is described in three progressively more complex versions (basic, intermediate, and detailed) in Boehm's Software Engineering Economics (3). In addition, for a nominal media charge, users can obtain a magnetic computer tape of the programmed model for installation and calibration at their own sites.

One of the hardest tasks in software development cost modeling is determining the extent of the effects of different attributes on development cost and schedule. Also, the interaction of these attributes is usually non-linear. In order to supply basic relationships between project attributes and productivity, Boehm employed both the analysis of project data and the results of a two-round wide-band Delphi technique administered to groups of software managers. In this way, a kind of expert system is used within COCOMO where the knowledge and experience of software managers is used to drive some of the calculations. This appears similar to PRICE S but the method of developing the data base appears to have been more formal.

Editorial

Probably the most important aspect of this model is that it is entirely open. The model is available on magnetic tape and Boehm's book is a more-than-complete guide to getting started with COCOMO. Interestingly, the complexity of the book has been known to discourage some potential users. This relates to the comments made by the marketing department at PRICE Systems who claimed that some of their customers prefer a black box to "do the work" for them. In the final analysis, however, it would appear the the effort to calibrate any of the models is comparable.

A brief interview with Don Alley, deputy to Wyn Royce, who is manager of Lockheed's Data Systems Engineering, a 700-person department, revealed that the COCOMO model has been in use there for about two years. It has been applied to about 20% of the on-going projects and all of the new "significant" programs (larger than $5M, according to Alley) since 1982. Often it is used on an on-going project when it deviates from schedule or when a general management tool is desired. The model has been judged successful at this environment. In the Fall of 1982, Lockheed used COCOMO to estimate not only their own part of MILSTAR, but also the sub-contracted pieces being developed by TRW and General Electric. (PRICE S was also used and was found to produce comparable results for the data provided.) As the three companies bid and scheduled the work, it became necessary for the sub-contractors also to acquire and use the model. Allen indicates that COCOMO provides more than just outputs for its users. It also forces its users to think through project parameters such as complexities and capabilities at various environments. In this way, a discipline of project analysis is developed as an important side effect.

SLIM

History

In 1977, or approximately one year after the first discussions of using the Rayleigh curve to depict software development effort, Putnam developed the basic software equation for this model while working at General Electric. After discussions within GE about the utility of the work, he left and started his own company (Quantitative Software Management) to develop and market a life cycle resource model for software development. This model is known as SLIM for Software LIfe-cycle Management model. A computerized version of the model was first available on a DEC 20 through American Management Time Sharing in 1979.

The model became available on micro computers with graphic capabilities in 1981. Recently a reliability model has been added to SLIM which computes error rate and error densities.

The cost is $20,000 per year for a DEC 20 license and $25,000 per year for the microcomputer version. This cost includes training and unlimited consultation. Additional sites within one organization are extra but pro-rated at a mutually agreeable discount. A bare bones implementation is available on a Hewlett Packard 75 pocket computer for $10,000, although none have been sold at this time. Also, a pocket calculator version was developed for demonstrations and in-house use but was never marketed.

Currently there are about 30 defense users and 30 commercial users including Hewlett Packard, GTE Data Systems, Hartford Insurance, Tektronix, and foreign users in the UK and France. The growth rate has recently been about 40% per year. With a data base of over 800 projects, Putnam expects to soon expand into productivity measurement and software measurement in general. The current thrust is to develop tools to automatically capture software development data.

Technical Summary

Although SLIM is a proprietary model, the underlying life cycle curve has been published by Putnam. It is generally regarded as a prescriptive tool, showing what should happen during development, and alerting managers when deviations occur. Nevertheless, it is not entirely a theoretical model, but rather a combination of an empirical model founded on theory of how problems are exposed and solved in a large development effort.

Editorial

Because of the inclusion of the Rayleigh-Norden theory of problem solving, SLIM differs from PRICE S and COCOMO which use mainly empirically derived data tuned by past experience at the user's organization to project cost. There are probably advantages to either approach (and the proponents of any of the models will generally explain these in some detail).

One comment from Quantitative Software Management was that the main impediment to more widespread use of SLIM or any formal cost model is the lack of sophistication on the part of most software development managers. It has been their experience that managers usually come for help when they are in trouble and it becomes obvious that something like this is important. Typically, at the time they come for help, the software developers are using a variety of informal techniques to plan a project, most often adopting the schedule specified by the customer or marketing department.

One common way for a software customer to take advantage of SLIM (or any model) is to require that all bidders submit historical data with their bid. In this way, the customer can use SLIM to verify that the bid is reasonable for that organization. This approach is recommended by Putnam over using the default values available in the model since these tend to produce conservative estimates.

IBM FSD

History

Although IBM Federal Systems Division, currently consisting of about 13,000 employees and 3,000 programmers and software managers, has never published a software cost model, some of the early work by Walston and Felix (9) provides some insight into their initial direction. A brief conversation with Claude Walston, now at ITT, provided some history about the data collection and modeling efforts there.

The internal software engineering effort was funded as a result of a 1971 business-level decision to quantify software production and to quantify the claims for modern programming practices, such as structured programming. The effort was aided by a simultaneous desire to unify and standardize on software development terms and measures across the division. Later, in 1971 the first locally derived data became available. The SDC work and other available studies at that time provided some guidance into the modeling and prediction process. The Software Engineering Cost Unit was formed, and with the support of a vice president, was able to obtain productivity and related data from the individual sites in FSD beginning in 1972. Although this unit was small, peaking at around 4 professionals, the support of the vice president ensured the necessary impetus for its success.

By 1974 several of the individual sites in FSD realized the advantages of having local software engineering cost functions, similar to their local hardware engineering cost functions. These local groups realized they needed even more data than that which was being requested by the central Software Engineering Cost Unit. As a policy, the central Software Engineering Cost Unit would only run their models on any new data after the local units had tried their own models. This was intended to encourage local accuracy and to discourage dependency on the central unit.

By the end of 1974, about 100 projects were providing data points for the calibration of the local and central models. Most estimates for proposed software development which were then funded, and subsequently tracked for validation of the models, were accurate to within 15% by this time.

Technical Summary

As far as is known outside the organization, no one particular model is used. Rather a combination of modeling and estimation techniques are made available to the managers.

Editorial

This brief view of IBM FSD was provided since it represents an important, if coveted, data point in the study of the emergence and popularity of cost modeling. It is unfortunate that so little has been published from this organization, but the pattern of internal development and internal use seems particularly comparable to COCOMO at TRW, although occurring a few years earlier.

Recap

This is a distillation of both the aspects that have facilitated and inhibited the use of the specific models discussed, as well as software modeling in general.

Inhibitors	Facilitators
PRICE S:	
Need to estimate machine code early in project. Closed model: start-up cost, minimal user sharing. Making revisions without impacting users is hard.	Established name with hardware model. Need to keep up with the capabilities of one's competition. Can be made as simple to use as the user perfers.
COCOMO:	
Apparent complexity. "Belongs to competition."	Open model. Ease of user sharing. Excellent user guidance available.
SLIM:	
Closed model: start-up cost, minimal user sharing.	Based on theory of how software development "ought" to proceed.
General: Evolving baselines. Needs both managerial direction and low level user support. Complexity: non-linearity of cause and effects, amount of experience required, time lag from decision to fruition. User sophistication slow. Results from models are most needed early, when they always provide their worst estimates.	ISPA (society, see PRICE, above). Many choices now available. Success stories Customer involvement. Contractor teaming makes model use contagious.

Overall Conclusions

In some ways, the evidence of "the long hard road" of technology transfer is not depicted in these accounts of the development of several of cost models. The four accounts here show about a three year lag between initial investment and substantial utility. (IBM FSD from 1971-74, COCOMO from 1976-78, SLIM from 1977-79, with PRICE S having some of its pre-1977 development subsumed by the PRICE H work.) There are at least two possible explanations for this apparent pattern. First, cost models appear to be evolutionary as opposed to revolutionary. They are not developed in a vacuum but rather build on work that has come before and attempt to improve on the shortcomings of their predecessors. (If it were necessary to cite some common point of departure for empirical modeling it would probably be the SDC data reported in 1966 (5). Although it may not be difficult to improve on that data, it remains one of the largest data bases even today, and at the time represented a pioneering effort in the field.) Therefore, each model attracts more followers to the field, though no single one can take credit for unilaterally advancing the state of the art.

The other explanation for the lack of evidence of the difficulty of inserting cost modeling into standard software development practice is that it just has not been inserted yet. Although the accounts provided here would suggest that an organization is likely to take stock in a locally derived model (or perhaps this is just the survival of the fittest, with the unfruitful efforts being cancelled), the development and use of formal models is not characteristic of the industry as a whole. Most organizations seem to be driven by budget and schedule which are set by the customer or by a market analysis. Most, however, have at least idealistic approaches to the organization of labor and tasks required to deliver a product. Although this understanding is the starting point for model development, the majority of organizations go no farther. DeMarco characterizes the typical software development estimate as "the most optimistic prediction that has a non-zero probability of coming true" (12).

An important aspect of using any of these models is experience. Since no two organizations respond exactly the same way to the various software development attributes, models must be calibrated to the user's organization. Similarly, a user must gain a feel for using a model and interpreting its output. For example, Park admits that the happiest users of PRICE S are the frequent users, while occasional users may be dissappointed.

Pointing to a fundamental problem with modeling software developments, Boehm suggests that the software development process is insufficiently understood to yield to rigorous statistical techniques (13). Techniques such as factor analysis and partial correlation will often yield insights but will probably not produce any true quantitative, replicable, and universal relationships given the current diversity of software development practices and environments. This is mainly due to the non-linearity of the cause and effect relationships which need to be quantified.

Empirical, algorithmic models suffer from certain weaknesses which probably provide some with sufficient excuses to avoid them. One obvious limitation is that their accuracy is constrained by the accuracy of the input data and also by that of the data used to calibrate them. However, Boehm states that the main difficulty is their inability to anticipate the effect of a base line change. For example, if all previous projects have been completed on one manufacturer's hardware, then it will be impossible to factor in the effect on productivity of using different hardware for the next project. This was experienced at the University of Maryland Software Engineering Laboratory immediately after the publication of the Meta Model (14). The next project used independent verification and validation for the first time at that organization. No previous observation could have been made as to the effect this would have on productivity and therefore the attribute "no IV&V" was built into the base line relationship of project size to cost. When the model was used to predict the cost of the project which used IV&V, it was in error by nearly 50% (15). To attempt to answer to this effect, most models provide defaults for the effect of various factors for which there is no experience at the environment under consideration.

Summary

The following aspects of the field of software process modeling are encouraging:

- Modeling has become a competitive field providing customers with substantial choices.

- At least one society, the International Society for Parametric Analysts is devoted to solving the problems inherent in modeling a complex human non-linear process.

- The utility can be felt in less than three years from the time an organization appropriates a useful amount of support for a modeling effort.

- Modeling such as this is an important pre-requisite to any further measurement an organization has decided to attempt; it is a first step into a quantitative self-evaluation.

The following aspects of the field are discouraging:

- To the novice, software modeling is a frustratingly complex and imprecise occupation.

- An organization that decides to shop for a model can be confused by the competition and associated advertising.

- The decision to perform data collection must be made at the level of the individual projects or it will not be done meaningfully, yet support for it must be demonstrated at higher corporate levels or the effort will not survive (16).

- The utility of a local modeling effort or metrics group will probably not be felt in the same budget year that the first efforts are funded.

Formal software development modeling is not a standard practice within the industry today. The recurring theme in discussing the spread of software cost modeling with various experts is that, in a contract world, it is often up to the customers to make the first move. By requiring historical data with bids or by announcing that a model will be used to develop the approved budget, a customer can sufficiently "raise the awareness" of (as in "leverage") the software developer. Therefore, in the final analysis, this must become a competitive requirement before the overhead will be invested on the part of all software vendors.

Appendix

Software Cost Modeling Chronology (17)

1956	-	H.D. Benington presentation, ONR Symposium. First reporting of large SW project costs.
1964-65	-	SDC regression study of 169 projects and their attributes.
1969	-	Aron model published.
1971-74	-	Internal IBM FSD data collection and model development.
1974	-	Wolverton model published.
1976-78	-	Internal COCOMO data analysis and model development.
1977	-	Doty model published.
	-	Walston and Felix publish some IBM FSD results.
	-	Boeing model published.
1977-79	-	SLIM developed and published.
	-	PRICE S developed and published.
1979	-	GRC models published.
	-	Albrecht publishes function point estimating techniques for early estimates.
1981	-	Bailey & Basili propose meta-modeling technique after no single model confirms NASA Goddard SE Lab data.
	-	COCOMO published.
	-	GRC model survey and validation finds none completely satisfactory.
	-	Dirks publishes Grumman model.
	-	Tausworth publishes JPL deep space model.
	-	Phister proposes theoretical model of software development based on programmer inter-communications and error detection.
	-	GSA software conversion cost model published by Houtz and Buschbach.

1982	-	Theoretical model of software project dynamics by Abdel-Hamid & Madnick.
	-	Simulation model of software life cycle by Duclos.
	-	Early sizing techniques demonstrated by Itakura and Takayanagi.
	-	Early sizing from state machine designs shown by Britcher and Gaffney.
	-	DeMarco proposes early sizing through structured analysis metrics.
1983	-	Jensen model published.
	-	SYSCON Corp. publishes a software support cost model.

Acknowledgements

Much of the information for this report was obtained through interviews of Robert Park, Larry Putnam, Doug Putnam, Barry Boehm, and Claude Walston. I am grateful for their assistance and patience.

References

(1) RCA PRICE Systems, "PRICE Software Model: Supplemental Information," RCA, Cherry Hill, NJ, March 1978.

(2) "SLIM System Description," Quantitative Software Management, Inc., McLean, Va., 1980.

(3) B.W. Boehm, Software Engineering Economics, Prentice Hall, Inc., Englewood Cliffs, N.J., 1981.

(4) R.W. Wolverton, "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, June 1974, pp. 615-636.

(5) E.A. Nelson, Management Handbook for the Estimation of Computer Programming Costs, AD-A648750, Systems Development Corp., October 31, 1966.

(6) J.D. Aron, Estimating Resources for Large Programming Systems, NATO Science Committee, Rome, Italy, October 1969.

(7) L.H. Putnam, "A Macro-Estimating Methodology for Software Development," Proceedings, Fall COMPCON 76, 13th IEEE Computer Society International Conference, September 1976, pp. 138-143.

(8) J.R. Herd, J.N. Postak, W.E. Russell, and K.R. Stewart, Software Cost Estimation Study - Study Results, Final Technical Report, RADC-TR-77-220, Vol. I, Doty Associates, Inc., Rockville, Md., June 1977.

(9) C.E. Walston and C.P. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, Vol. 16, No. 1, 1977, pp. 54-73.

(10) R.K.D. Black, R.P. Curnow, R. Katz, and M.D. Gray, BCS Software Production Data, Final Technical Report, RADC-TR-77-116, Boeing Computer Services, Inc., March 1977.

(11) L.H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering, July 1978, pp. 345-361.

(12) T. DeMarco, Controlling Software Projects, Yourdon Press, New York, 1982, p. 14.

(13) B.W. Boehm, op. cit.

(14) J.W. Bailey and V.R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering, March, 1981, pp. 107-116.

(15) Personal experience, and communication with Dr. Gerald Page of Computer Sciences Corporation.

(16) This conclusion was extracted both from personal experience and from discussions with Claude Walston who has worked at both IBM and at ITT, which have different styles of corporate management and different levels of policy-making affecting the software development components.

(17) For additional chronology, see also B.W. Boehm, "Software Engineering Economics," IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, January 1984, pp. 4-21.

Case Studies of Software Engineering Technology Transfer

Technical Memorandum

Paul C. Clements
Lou Chmura
Stuart Faulk
Preston Mullen
Alan Parker
David Weiss

Naval Research Laboratory

April 1984

INTRODUCTION

This memorandum describes several examples of the transfer of the software engineering technology developed and applied in the Naval Research Laboratory's Software Cost Reduction (SCR) project, in which modern software engineering techniques are being used to specify, design, and build a model system so that the techniques can be emulated by others. The system chosen as a model is real and complex -- the onboard flight software for the Navy's A-7E aircraft. The principal techniques used include a new method for specifying software requirements, a method of module decomposition and design using information-hiding, a method for building the system in increments by specifying the "uses" hierarchy, and a method of using cooperating sequential processes to implement the run-time structure of the system.

Each case study consists of sections explaining the application area in which particular techniques are being used, the technology that was transferred, and the technology transfer process. Aids and obstacles to the transfer process are discussed in each case, as well as the effects of the technology in its new application.

The case studies are given in the next section. The last section reports the conclusions from the SCR experience in technology transfer and suggests ways in which technology insertion may be made more effective.

The SCR project is a significant step in the transfer of modern software engineering technology into the programming workplace. It breaks many of the software bottlenecks described by Graham (Gra82) by providing a complete, integrated software development methodology. Evidence cited by Graham indicates that 20 years is the typical delay between the time a laboratory prototype is built and the time the use of the innovation becomes widespread. SCR is reducing this delay time for software engineering technology by combining a number of techniques that have been in the laboratory prototype stage into a working model that can be distributed and used widely. As is shown in the case studies following, this is achieved by publication of SCR documents in the open scientific literature, by publication and presentation of technical papers describing the SCR technology, and by the provision of consulting services by the SCR project staff to others interested in using the technology.

Introduction References

(Gra82) Graham, Alan K.; Software design: breaking the bottleneck, IEEE Spectrum, pp 43-50, March 1982

Case History: Bell-Northern

1. Application

SCR software design technology has been used in the development of the DMS 100 (Digital Multiplex System) family of digital telephone switching systems developed by Bell-Northern Research Ltd., in cooperation with Bell Canada and Northern Telecom (1). DMS is a large, software-controlled real-time switching system. The software implements a diverse set of telephone functions as well as systems for administration and maintenance.

2. Technology Adopted

The DMS 100 system design philosophy is the same as that of the SCR -- modularity, hierarchical design and information hiding. Different telephone companies require different software features; hence, the system must be easily configurable to implement a variety of slightly differing software packages. In addition, the software must be frequently changed to add new features or correct errors. The concepts of modularity and information hiding are used to limit the effects of changes or corrections. The system is designed as a "uses" hierarchy of modules, allowing a variety of slightly different systems to be easily generated by changing the subset of modules implementing a given system.

3. How Technology Was Transferred

The design methodology was derived by the personnel of Bell-Northern Research from the SCR methodology that is described in (2), (3), (4) and (5). According to the project director, prior experience with other methodologies had convinced the development team of the usefulness of advanced software engineering techniques in general and the methods proposed by Dr. Parnas in particular. The team developed a language (PROTEL), linker, and library maintenance system to support the methodology. With these tools, the methodology proved natural and easy to use.

4. **Obstacles to Transfer**

None.

5. **Aids to Transfer**

None.

6. **Results**

The DMS 100 family of systems is currently operational and is being successfully marketed by the parent company. According to the project director, since the release of the DMS 100 system, Bell-Northern has become the largest supplier of such telephone switching systems in the United States. He attributes this success to the ease of configurability and maintenance of the DMS 100 software. An overview of the system and the design methodology employed are given in (1).

References

(1) Lasker, D.M., "Module Structure in an Evolving Family of Real Time Systems", Proc. Fourth Intl. Conf. on Software Engineering (September 1979), pp. 22-28.

(2) Parnas, D.L., "Designing Software for Ease of Extension and Contraction", Proc. Third Intl. Conf. on Software Engineering (May 1978).

(3) Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", Comm. ACM, Vol. 15, No. 12, pp. 1053-1058 (December 1972).

(4) Parnas, D.L., "On a 'Buzzword': Hierarchical Structure", Proc. IFIP 1974 (1974).

(5) Parnas, D.L., and Wuerges, H., "Response to Undesired Events in Software Systems", Proc. Second Intl. Conf. on Software Engineering, pp. 437-446 (October 1976).

Case History: Bell Labs

1. Application

Starting in 1978, AT&T Bell Laboratories (BTL) in Columbus, Ohio, began using various portions of the SCR methodology in their system development process. The first project to use these techniques was the No. 2 Service Evaluation System (SES), which is a multiprocessor data acquisition and transaction system for telephone network quality control. The project took place between 1978 and early 1981. The development team consisted of approximately 10 persons.

2. Technology Adopted

Most of the SCR methods for requirements specification and system design were adapted for use in this project. A paper (Hester) published in 1981 describes the methods as they were applied to No. 2 SES.

In the requirements specification, the techniques used for the A-7 were modified to suit the "transaction-oriented nature of the No. 2 SES". Performance requirements, responses to undesired events, assumptions about changes, required subsets, and glossary are dealt with essentially as in (REQ). The section on modes is absent. In general, formalisms are simpler than in (REQ). Other aspects of the organization of (REQ) are modified as noted below:

In the section on data items, data types are introduced to describe data items. Data type names are bracketed +...+. The "intermediate data item" (the output of one function used as input to another, but not directly visible to user) is introduced and used to facilitate breaking up functions into subfunctions that are likely to change separately.

A new section on communications protocols was added, documenting system interaction with external hardware, external software systems, and users (command syntax). It seems to generalize information from (REQ) section 1 (computer characteristics) and 2 (hardware data items). The section on user transactions and reports corresponds to the "functions" of (REQ) and defines user-visible functions in terms of data items. User transactions allow users to parameterize requests, such as report generation commands. Other functions include "computer operations, data base interactions, ... maintenance, and spontaneously generated reports". Event tables as in (REQ) are not used.

Other documents based on those of the SCR project include the module decomposition document (based on the A-7 module guide (MG)) and individual module interface specifications.

3. <u>How Technology Was Transferred</u>

The technology was transferred in the following ways:

(1) A full-time technology transfer agent position was created within BTL. The agent was responsible for recommending technology to be transferred. He was available to discuss the technology and to help projects start to use the technology. He educated himself in the SCR technology through perusal of the available technical literature and through infrequent, informal contacts with SCR project personnel. He also arranged for a visit to BTL by the principal investigator of SCR, Dr. David Parnas (see next item).

(2) Dr. Parnas spent a short period of time (less than 2 weeks) at BTL describing the technology and discussing it with potential users.

(3) BTL project personnel used SCR documentation, published by NRL and in technical journals and conference proceedings, as models on which to base their use of the technology.

4. <u>Obstacles to Transfer</u>

The No. 2 SES project was handicapped by adopting the SCR principles after the project was underway, making no allowances in scheduling or staffing; the principles had to be learned and tuned even as they were being used in system development. In addition, it appears that some members of the small team may have resisted generating so much documentation, feeling that they had already mastered the requirements and design decisions.

5. Aids to Transfer

Technology transfer in this case was aided by the project leader's belief in the SCR methods based on publications in the professional literature and by the existence of the BTL technology transfer agent whose responsibilities included learning and understanding new software engineering technology.

A major aid in the No. 2 SES came when the SCR project's principal investigator and the originator of many of the fundamental SCR methods was retained as a part-time consultant during the critical time when the methods were being adopted.

6. Results

The No. 2 SES was deployed on schedule in 1981. The extra time invested in thoroughly documenting requirements and interfaces was recouped during the exceptionally smooth system integration.

As a result of the success of this initial experience with these methods, BTL are now using them in a number of projects of various kinds. To quote from (Utter),

> The usage curve has been upward; three times as many projects have adopted it in the past year as in the first two. Every project has continued to use it. All the use has been done on development projects within existing schedules, with no specific training.

The publication of (Hester) in the Bell System Technical Journal and the existence of worked-out examples has increased the spread of these techniques in the organization. Project managers can see how successful the tech-niques have been for others, and documents from existing projects (especially the SCR project and the No. 2 SES) can be used as paradigms for new projects. One recent figure from BTL indicates that 6 requirements specifications using these techniques have already been completed and another 16 are in progress. Within AT&T Bell Laboratories the techniques are now known by the title "System Design Through Documentation", or SDTD.

Unfortunately, detailed information on individual projects (such as copies of requirements or design documents) or size of the completed systems is not available, because BTL consider the information proprietary. Similarly, there is no good detailed public data on the extent of usage. Nevertheless, it is clear that software developers at BTL have been able to use successfully the underlying SCR methods based on information hiding, separation of concerns, and disciplined documentation, modifying them when necessary to fit their own projects. Furthermore, the same set of modified techniques has sufficed for various kinds of projects; it is not necessary to reinvent the techniques for every application.

References

(Hester) S.D. Hester, D.L. Parnas, D.F. Utter, "Using Documentation as a Software Design Medium." Bell System Technical Journal, Vol. 60, No. 8, October 1981, pp. 1941-1977.

(Utter) D.F. Utter, "Properties of the System Design Through Documentation (SDTD) Methodology".

(REQ) Heninger, K.L., Parker, R.A., Parnas, D.L, Shore, J.E, Software Requirements for the A-7E Aircraft; NRL Memorandum Report 3876; November 1978.

(MG) Britton, K.H., Parnas, D.L., A7E Software Module Guide; NRL Memorandum Report 4702; December 1981.

Case History: Softech, Inc.

1. Application

The DWS/CS Emergency Preset (EP) weapon control system was designed to be installed aboard Trident submarines as a backup to the primary fire control system. The overall system requirements call for a system based on a single processor and software of "modest" size (under 100,000 instructions), whose operation and interfaces would be compatible with those of the main fire control system (known as DWS/CS). Project management wished to avoid problems experienced with the original DWS/CS: difficulty of use (and of training users), expense of change, long familiarization time for software maintainers, and lack of a clear and definitive specification.

2. Technology Adopted

Under NUSC and NAVSEA sponsorship, Softech, Inc., produced an EP software requirements document modeled on SCR's A-7 requirements document (REQ). The EP document completed in 1982 follows (REQ) closely in most respects. The exceptions are described below.

The most obvious departure from (REQ) is the use of Softech's proprietary Structured Analysis and Design Technique (SADT). This graphical aid is used in two ways. First, SADT was used as a systems analysis tool to generate "activity models" of the EP system and its environment; the EP requirements were then analyzed using these models. The developers found this necessary because, unlike the A-7 project, which used the existing OFP as the basis for the detailed software requirements, there was no existing model from which to identify the detailed EP requirements and, indeed, no clear understanding of them.

The second use of SADT was to model the operating modes and

mode transitions of the system. These "mode models" served as a "usual case" abstraction of the normal sequence of system operation and formed the basis of the mode tables in the style of (REQ); the mode tables were then filled out with transitions handling all the unusual mode transitions that could also arise. It should be stressed that the standard mode tables do stand alone as a
definitive reference; although they contributed to the development of the detailed requirements, the SADT drawings now serve only as introductory overview material.

The other departures from (REQ) are extensions of existing concepts. For instance, to specify the behavior of a general-purpose display device, the EP requirements formalizes the notion of a "semantic entity", i.e., a logical output datum considered without regard to the conditions determining when and in what format it is sent to an output device. The document then specifies a set of functions that calculate these semantic entities, a set of "display" functions that display certain semantic entities on the display console, and a set of "set and transmit" functions that send certain semantic entities to weapon system hardware.

In a minor modification to the (REQ) nomenclature, the concepts of "mode" and "stage" are unified in a generalized mode that can have "subordinate" modes. As usual, a mode is simply represented by a set of conditions; if a mode has subordinate modes, then its conditions must hold in each of the subordinate modes, with the subordinates being distinguished from one another by additional conditions.

3. <u>How Technology Was Transferred</u>

The decision to adopt the SCR Requirements format was made independently by the contractor on the basis of the published A-7E Software Requirements document and the contractor's ongoing evaluation of software engineering methods. The contractor felt that the SCR methods would help avoid the problems, mentioned above, that had plagued the original DWS/CS system. There was no contact between SCR personnel and the contractor, nor was any incentive paid to the contractor by the sponsor to encourage the use of SCR techniques. The cost of the technology transfer was simply the time and effort incurred by the contractor to learn and, where necessary, extend the SCR techniques.

4. Obstacles to Transfer

The major obstacle to technology transfer was the hesitancy of some contractor personnel to try out unfamiliar methods, and an initial inclination to modify the methods unnecessarily. As an example, there was considerable discussion in the early stages of the project concerning whether the SCR notation should be retained or replaced. Some time was spent in developing and then discarding replacement notation.

5. Aids to Transfer

Aside from the existence of the published SCR requirements document and associated descriptive papers, no specific aids for transferring the technology were used. There was no contact between the SCR project and either Softech or NUSC. But there was contact between Softech and another contractor who had experience with the SCR methods.

6. Results

The contractor produced a complete detailed software requirements specification for the EP system, in the manner of (REQ). The contractor was able to apply the SCR methods in a straightforward manner, to modify them independently and sensibly, and to integrate them usefully with the contractor's own proprietary techniques. Software engineers at all skill levels were involved in this effort and most had no trouble with the SCR methods.

Although the EP requirements were completed, the Navy decided that the backup system was not needed aboard the submarine, so funding for the EP system was canceled. Thus, there has been no chance to use these requirements as a basis for further system development.

References

(REQ)   Heninger, K.L., Parker, R.A., Parnas, D.L., Shore, J.E.,; Software Requirements for the A7E Aircraft; NRL Memorandum Report 3876; November 1978.

Case History: Naval Weapons Center A-7E Program

1. Application

The responsibility for maintaining and updating the software for the A-7E aircraft rests with the Naval Weapons Center in China Lake, California. The Operational Flight Program (OFP) in the aircraft resides in a small and rather archaic embedded computer. The program controls instruments and indicators, and receives information from sensors and other equipment. Navigation instruments include an inertial measurement set, Doppler radar, and atmospheric sensors. Navigation information is displayed on a digital panel, or by moving a projected color map of the overflown terrain. Attack equipment includes a forward-looking radar, forward-looking infrared, and a repertoire of almost 100 kinds of weapons. Attack information is shown on a head-up display. Target location is communicated to the program by keyboard entry, or by "pointing" to the location on one of the displays (e.g., overlaying the location with a cursor on the radar screen). The program then computes when to release a chosen weapon, provides flying cues to the pilot, and if desired will actually launch the weapon at the most favorable time. The release calculations are quite complex, involving the flight and ballistic characteristics of the particular weapon being used.

The A-7E OFP is a program that meets very complex requirements, running under very tight time and space constraints. Even so, it provides capability not found in other aircraft systems. For instance, the A-6 lacks a map display; nor does it offer the attack mode most preferred by pilots, in which the impact point of a weapon is continuously computed. Further, the A-7E OFP has the reputation of being one of the most reliable and defect-free programs in the Navy.

2. Technology Adopted

An NRL-style requirements document has been written for the version of OFP known as NWC-4. That document has also been updated to capture the requirements of the first (of four) incremental changes that will become NWC-5. Further, the NRL requirements document that specified the requirements for NWC-2 has been modified and used as a baseline to derive the software validation procedures for NWC-2F,WC-4, HAF-2 (for A-7s in the Hellenic Air Force of Greece), and NWC-5.

3. How Technology Was Transferred

Since the demonstration vehicle for NRL's Software Cost Reduction project is the A-7E flight software, NRL personnel have had extensive contact with the NWC A-7 staff. SCR's first product was a complete requirements specification of NWC-2. Since the A-7 software requirements had never before been written down in one document, copies of the NRL document soon began appearing on desks at NWC. By the time that NWC-4 was on the horizon (eight versions later), the NRL document was trusted and understood well enough to be accepted.

4. Obstacles to Transfer

There was some concern that the NRL-style document might not satisfy the applicable MIL standard. However, NWC took the approach that the document clearly satisfied the intent, if not the organizational letter, of the standard.

5. Aids to Transfer

The management of the A-7 project office at NWC was open to new ideas and trusted the people in the software branch when they said that this would help them to turn out a better product. The previous lack of such a document was also a key factor. Finally, the fact that creating such a document would be a matter of updating the NRL document (rather than starting from scratch) certainly didn't hurt.

6. Results

The NWC-4 document was produced for under $200,000, which includes data entry, as well as labor cost during the learning phase of the project. The A-7 also has "better validation procedures than we've ever had before" in the opinion of a member of the NWC staff, who also thinks that A-7 "has the most complete (software) documentation of any aircraft project at NWC."

They also probably have the most maintainable. Previously, software requirements were scattered throughout several documents, including a NATOPS manual for the aircraft ("NATOPS"), a tactical supplement to the NATOPS ("TAC"), and a maintenance instruction manual ("MIMS") for shipboard maintenance of the aircraft. Below is a table showing the cost (in dollars and pages) of changing these manuals for two recent software modifications. Note the contrast with NWC's software requirements document ("SRD").

	NATOPS	TAC	MIMS	SRD
Change A	$3300	$10500	$17500	$400
	12 pages	36 pages	64 pages	2 pages
Change B	$3300	$110000	$309000	$400
	12 pages	37pages	1131 pages	2 pages

Another estimate of maintainability is that NWC is planning to devote only one person working half-time for one year to change the NWC-4 requirements to NWC-5.

Case History: Naval Weapons Center A-6E Program

1. Application

The A-6E Intruder aircraft is a carrier-based two-man medium attack aircraft. Its operational flight program (OFP) is very similar in nature to that of the A-7E in mission, size, and complexity.

2. Technology Adopted

For the next version of the A-6E software, NWC China Lake is writing an NRL-style software requirements document.

3. How Technology Was Transferred

Motivation to use the NRL methodology was provided primarily by the NWC A-7 experience. A-7 personnel provided much encouragement and performed what can be fairly called a lobbying effort to get the appropriate A-6 people to recognize the benefits of the technology. A-6 management tends to be more conservative and less inclined to depart from the more traditional style of managing software. If the A-7 experience had not been close at hand, it seems clear that the A-6 effort would not have been launched.

4. Obstacles to Transfer

The people initially assigned to write the document did not have a full understanding of the technique, and hence produced little of value for a long time. This had the effect of tarnishing the methodology in the eyes of an already-reluctant A-6 management team. Management reluctance to depart from safe traditional methods also hindered the effort. There was also a certain tendency to unnecessarily modify the methods by a subsequent contractor, which resulted in a couple of false starts. Finally, the A-6 management became concerned when prolonged discussions about the requirements became commonplace. However, this was turned into an advantage when it was pointed out that the discussions were revolving about what the A-6 software requirements actually were, rather than about how to use the NRL technique to specify them. We were able to point out that because of the methodology, ambiguous requirements were being discovered very early in the development process, at a point when they were the least expensive to resolve.

5. Aids to Transfer

The willingness of both NRL and NWC A-7 personnel to consult and review played a major role. The clear superiority of the methodology over anything previously used was also a key factor.

6. Results

The document is still in preparation. NRL personnel have attended briefings and provided reviews, and the document appears to be well under way on a successful track. The A-6 management appears committed to the methodology, and have contracted with Grumman Aerospace Corporation to write a manual describing how to produce an NRL-style requirements document.

Case History: USAF A-7D

1. Application

The USAF also flies the A-7 aircraft; its program is very similar to the Navy software, except that the class of available weapons is different, and it does not include facilities for shipboard inertial alignment. The computer used is the same as for the Navy A-7.

2. Technology Adopted

An NRL-style requirements document has been written for the USAF A-7D software. In addition to completely and accurately describing the current program, the document is used to specify changes to the software. ~~When a change is needed,~~ it is specified by changing the appropriate page or pages in the software requirements document, and then presenting those new pages to the contractor with the instructions to make the program meet those new requirements.

3. How Technology Was Transferred

As might be expected, the USAF A-7 office and the NWC A-7 office maintain close contact. Many changes to one program are also desirable to the other (e.g., the introduction of a more accurate ballistic algorithm). Thus, when NRL began working with NWC, we also became acquainted with the USAF A-7 personnel. The major vehicle of technology transfer in this case was an intensive two-week course in software engineering principles that NRL presents from time to time. The course was attended by key USAF A-7 personnel who recognized the potential benefits of the methodology to their application. They produced the A-7D software requirements document entirely on their own initiative.

4. Obstacles to Transfer

None.

5. Aids to Transfer

The person in charge of the A-7D software is an individual involved in the technical as well as management issues of his project. Thus, he could immediately recognize the advantages of the methodology, and was in a position to choose the course of his project. And as with NWC, he only had to update the NRL document, rather than start from scratch.

6. Results

The Air Force has, for the first time, a document which completely describes the software for their A-7 aircraft. Using the requirements to specify changes also gives them a powerful contractual tool. Because the requirements (and hence the software changes) are specified completely and unambiguously, contractor performance can be stringently tested.

Case History: Requirements Specification at Tektronix Corp.

1. Application

An applications group at Tektronix Corp. is currently using techniques developed by the SCR project to develop the software for a new graphics input product (details of the application are a trade secret at this time). The application is based on dual microprocessors (Motorola 68000) operating concurrently. The application is control intensive and interrupt driven. Besides responding to external interrupts and controlling devices (motors), the system software is responsible for data acquisition and computations on the data.

2. Technology Adopted

The Tektronix group has adopted the system requirements methodology developed by the SCR project. With the addition of some data flow techniques, the methodology has been used to write all of the system requirements. In addition, the SCR supported design techniques of modularity, information hiding and a "uses" hierarchy have been applied in the design and development of the system software.

3. How Technology Was Transferred

The use of the SCR Requirements methodology has been adopted by the Tektronix team at the instigation of the project supervisor. The supervisor became interested in the SCR methodology following a presentation by one of the SCR staff at the Software Reliability Conference in Boston (1982). Subsequently, he attended a week long lecture course given by Dr. Parnas and has received all of the SCR publications. The supervisor has maintained contact with the SCR project through Dr. Parnas who has made himself available to answer questions concerning the methodology.

The project supervisor reports that although he was given primary responsibility for choosing his own development methodology, he has found the SCR Requirements document and other NRL publications useful in justifying his approach to his supervisors and team personnel. He also reports that most of his programming team (four out of five) have received the new methodology enthusiastically and had little trouble using the SCR techniques.

4. **Obstacles to Transfer**

None.

5. **Aids to Transfer**

Availability of published NRL documents in the literature.

6. **Results**

At this time, all of the system requirements have been written using the SCR requirements methodology. The system has been designed and written using the aforementioned techniques of modularity and info~~r~~mation hiding, and the system is currently being tested and debugged. The team supervisor reports that the use of the SCR methodology has resulted in a requirements document of high quality and that the document has proven extremely useful throughout subsequent phases of the development. In addition, the team has found the underlying principles supported by the SCR methodology (e.g., information hiding) useful throughout the design phase for making crucial design decisions.

Case History: RFP for New Class VI Computer Application

1. Application

NRL has released an RFP for a new class VI computer system. This system will comprise a high power vector cpu as a back-end and a front-end processor providing functions such as communications, local editing, and mass storage. The function of interest in this report is communications, specifically communications with the DoD Internet (Arpanet and Milnet) and the Lab's TCP/IP based local network.

The problem was how to specify the interface to the networks. The interface to Milnet could be completely specified, because the hardware is in place. But it would be desirable to design the system so that changing the Milnet hardware would not cause major changes in the rest of the system. The connection to the NRL local network couldn't be completely specified, because at the time of the RFP it wasn't known exactly what hardware would be used. It was known that it would use the DoD TCP/IP protocols. The solution to both of these problems was to design an abstract interface for the network interface device.

2. Technology Adopted

An abstract interface for two modules was included in the RFP. One abstract interface was for the data link device module. This module is to contain the device dependent details of the particular network interface involved. For example, one module might know details of the BBN C/30 IMP, another might know details of an Ethernet controller. The other abstract interface is for an address mapping module. This is required to map from Internet addresses to the addresses on a particular physical network. The form of addresses on a particular network is a characteristic of that network, and should not be known throughout the system. Outside of this module, only Internet addresses are used. These interfaces are described in detail by reference (1), which became a part of the RFP for the computer system.

3. How Technology Was Transferred

SCR personnel participate in a committee that is designing a Lab wide network at NRL. During a committee meeting (also attended by people from the class VI computer RFP group) mention was made of the difficulty is specifying the network hardware/ software interface in the RFP. During a later review of the RFP, an abstract interface to describe the required interfaces was suggested. SCR personnel designed the interfaces and prepared a Technical Memorandum to be included in the RFP. This involved about 0.5 man month.

4. Obstacles to Transfer

Some people questioned whether such a thing could be included in an RFP, mainly because it had never been done before. The committee probably didn't quite understand what was being attempted.

5. Aids to Transfer

The top-level management in this case is familiar with the technology and brought pressure to bear on the committee to accept the ideas.

6. Results

The abstract interfaces were included in the official RFP. They will force the bidders to modularize the network implementation in such a way that network hardware changes should not be hard to handle. They also provided a way to specify all that was known at the time the RFP was written concerning the required network interconnections.

References

(1) An Abstract Interface Specification for the Data Link Device Module and the Address Mapping Module for a Network Using the DoD Internet Protocols, NRL Tech Memo 7590-233, 19 Sept. 1983.

Case History: NRL Architecture Research Facility

1. Application

NRL designed and implemented a large FORTRAN program called the Architecture Research Facility (ARF). ARF is a program that when given a formal description of a computer instruction set architecture provides simulation of that architecture. The system was used in support of the joint Army/Navy Military Computer Family (MCF) project.

2. Technology Adopted

The primary goal of the ARF designers was to produce a working simulator; certain software engineering principles were employed to facilitate this. The ARF was to be developed using the family approach to software development. The information hiding principle was to be applied to conceal design decisions that were expected to change during the lifetime of the ARF. Several debugging aids were designed into the system to make development easier. These included:

- A method for detecting errors involving improper access to table entries.
- A consistent execution-time error reporting scheme for table interface functions that preserved the name of the routine in which the error occurred and reported a code associated with the error.
- A mechanism for inserting, and turning on and off, debugging code through the use of a compile-time preprocessor.

3. How Technology Was Transferred

Most of the technology was spawned from within, since all work was done at NRL. At the time, most of the software engineering principles had appeared in the literature, and Dr. David Parnas was a consultant to NRL.

4. <u>**Obstacles to Transfer**</u>

None.

5. <u>**Aids to Transfer**</u>

None.

6. <u>**Results**</u>

Although ARF was primarily a research project, it was completed and it worked well. It was extensively used by NRL to develop a formal description of one of the candidate MCF architectures. ARF is remembered at NRL as the project that laid the groundwork for the Software Cost Reduction project, because it was one of the first systems to apply certain software engineering principles to a real problem. NRL's first experience with such principles came from ARF.

References

(1) Evaluating Software Development by Error Analysis: The Data From the Architecture Research Facility; David M. Weiss, NRL Report 8268, December 22, 1978;

(2) Architecture Research Facility: An Experiment in Software Engineering; Honey S. Elovitz, NRL Report 8346, December 31, 1979.

Case History: Advanced Narrowband Digital Voice Terminal (ANDVT)

1. **Application**

The Advanced Narrowband Digital Voice Terminal (ANDVT) is a TRI-TAC program to provide secure voice and secure data communication facilities to users with only narrowband (Approximately 3KHz) channel capability. The first equipment developed under the program is a tactical terminal (TACTERM) designed for shipboard, airborne, shelter, and vehicle applications over HF radio, fieldwire, troposcatter, and Line of Sight (LOS) communication circuits. The TACTERM development has been a joint effort of the Naval Electronic Systems Command (NAVELEX) and the National Security Agency (NSA). The Naval Research Laboratory (NRL) has served as the NAVELEX technical agent and has performed the system engineering function for the development.

The schedule for the ANDVT TACTERM development is as follows:

Milestone	Date(s)
Conceptual Phase	Sep 76 - Oct 78
Feasibility Phase	Oct 78 - Sep 80
Full-Scale Engineering Development Phase	Oct 80 - Mar 84
Initial Production Phase	Jan 84
Initial Operational Capability	Oct 87

2. **Technology Adopted**

The major technology applied to the TACTERM development was that identified in MIL-STD-1679, which was being written during planning for the TACTERM Feasibility Phase. MIL-STD-1679 work and data requirements were written into the Feasibility Phase Request For Proposal.

3. How Technology Was Transferred

Because of the newness of the MIL-STD-1679 methodology, the NRL system engineering team sought help from the Computer Science and Systems Branch at NRL and obtained a limited amount of consulting
assistance from two software engineers. Nothing that may be called truly innovative software development technology was used in the TACTERM development; superior software documentation was not produced. The major contribution of the software consultants was that software issues were raised early on in the development and the contractor was forced to deal with them. For example, the engineers (1) evaluated proposals for the Feasibility Phase contracts; (2) forced comprehensive Software Development Plans; (3) caused a lot of attention to be paid to Program Performance Specifications and, as a result, forced beneficial communication among the contractor's system engineers and software engineers; and (4) generally caused contractor personnel to pay careful attention to delivering quality software documentation.

4. Obstacles to Transfer

None.

5. Aids to Transfer

None.

6. Results

The NRL system engineering team has observed that software problems have been few and those that have occurred have been relatively straightforward to correct. The conclusion is that the technology when supported by part-time consulting can reap benefits. If, however, truly innovative development approaches or products are desired, then the approaches and products must be defined and pursued early on.

Case History: Training Integration System

1. Application

The T-45TS is to be the Navy's replacement for the undergraduate jet flight training system. Overall, the new system will include an aircraft, simulators, and a Training Integration System (TIS). The TIS subsystem is envisioned as a large software system that will be used in such diverse aspects of training management as personnel records, financial information, and, of major importance, daily and long term scheduling of training resources. Critical training resources such as airplanes, simulators, and instructors must be matched up with students in the most efficient manner that will ensure timely graduation. Thus, while most of the TIS functions appear to be typical management-oriented database applications, the addition of the scheduling problem and the requirement for near-real-time response to changes in resource availability make the system more difficult to specify.

Based on a competition in which several proposals were entered, the Navy selected the Douglas Aircraft Company as the prime contractor for the T-45TS. The simulators and most aspects of the aircraft are to be subcontracted, but Douglas will implement the TIS in-house.

2. Technology Adopted

NRL personnel consulted for NAVAIR on the TIS subsystem specification. The project had not reached the point where any but the most general requirements could be ascertained. It was therefore inappropriate to develop a specification with the sort of detail found in the model requirements document (REQ). The main effect of NRL assistance was the adoption of a more systematic classification of system functions (although the list was still not very detailed) and the inclusion of requirements that, it was hoped, would have the effect of making the resulting system easier to change.

3. How Technology Was Transferred

NRL personnel were engaged to consult on the development of the TIS specification by the NAVAIR command responsible for the T-45TS. Consulting was a part-time assignment for two people.

4. Obstacles to Transfer

Initial meetings attended by NRL personnel were Navy-only and focussed on identification of system requirements and approaches to the system specification. At first, the Navy project managers showed tendencies to write a "kitchen sink" specification (e.g., "The system will do everything necessary to support training.") and to assume that the contractor could be trusted to make everything right. However, NRL personnel explained why this was inadvisable, and gradually the situation changed.

Later meetings included contractor representatives and focussed on ironing out problems in the system specification. Most of this time was spent over contractual issues having nothing to do with software engineering. When software matters were addressed, the "old guard" attitudes and the lack of technical expertise of the contractor personnel made these discussions difficult. The difficulty of getting the contractor to understand ease of change as a criterion for the TIS software can best be illustrated by a remark made by one of the contractor's management representatives: "If you want a change in the system, no problem: just pay us and we'll change it." The contractor's personnel were, for the most part, entrenched in the familiar way of arranging such contracts, and when contractor personnel did show an understanding of the NRL approach, they were invariably lower-level employees without power to influence things.

5. Aids to Transfer

The steadfast support of the NAVAIR project managers enabled the NRL people to weather the opposition of personnel from other

Navy agencies and eventually to gain their agreement.

6. Results

The system specification for the TIS is more detailed and better organized than it would have been, but it still offers too little detail on which to base contract pricing, and it does not provide a detailed software performance specification.

References

(REQ) Heninger, K.L., Parker, R.A., Parnas, D.L, Shore, J.E; Software Requirements for the A-7E

Case History: Collection System (CS)

1. Application

The Collection System (CS) is a family of systems each of which has to track and command, as well as gather and preprocess, data from satellites. It replaces an existing family of systems with similar, but outdated, capabilities. The CS developmental life cycle (i.e., the time from initial concept to deployment of the final site) covers approximately eight years, the last five years of which might be called a full-scale engineering development (FSED) phase. Some FSED-phase milestones are as follows:

Milestone	Month
System Requirements Review	5
System Design Review	14
Preliminary Design Review	21
Critical Design Review	28
Production Release	39
Development Test and Evaluation	46
All sites operational	61

The CS development is a multimillion dollar effort. During the FSED phase, the manpower effort averages 200 man years per year. Because the project is classified, additional information is not available.

2. Technology Adopted

New technologies are being applied in the CS development at the instigation of the project manager, who was introduced to many of the technologies in two software engineering courses offered by the Computer Science and Systems Branch at NRL. The new acquisition-management technology being applied in the CS development follows current MIL-STDs applicable to hardware and software developments. The new engineering technology being applied comprises much of what is being applied and refined in the STARS-funded Software Cost Reduction (SCR) project.

3. How Technology Was Transferred

Three engineers from the Computer Science and Systems Branch at NRL (two of whom also work on the SCR project) work part time on the CS development and provide the primary assistance to full-time project engineers in applying the new technologies. There is some consulting with SCR project personnel on a free-of-charge, as-needed basis. Contracts are sometimes issued to acquire expert support for particular technologies.

4. Obstacles to Transfer

There are two obstacles to the transfer; they concern psychological aspects of DoD system developments. First, because of the natural human resistance to change, it is difficult to introduce new technology. The difficulty is not limited to contractor personnel; it extends to government personnel working on the project. (Difficulty with a contractor may be avoided to some extent by making things clear in the Request For Proposal.) Second, because some new technologies emphasize early project tasks and products, they delay traditional milestones (e.g., the production of code or software). Because of this and because most DoD developments are pressed for time and money, the result is that people become nervous about the undertaking and resist the introduction of the technology.

5. Aids to Transfer

None.

6. Results

The project is still in the early stages. The SCR techniques are being used by the government and contractor personnel. Concerning the use of the SCR project's formal techniques for specifying system (specifically software) requirements, it appears
that people feel the need to first document the results of system analysis using traditional English or graphical approaches. They then find it easier to translate these informal specifications into the formal SCR software requirements document than to generate the formal requirements from scratch. This agrees with the findings of the NUSC-funded effort to specify requirements for the Trident's DWS/CS Emergency Preset (EP) Subsystem.

Case History: A Satellite Communications System

1. Application

The system will provide a tri-service satellite communications system. One particular part of the project is to produce the ground terminal and the associated communication functions -- antenna controls, modems, multiplexers, encryption devices, etc. The terminal design must be applicable for the communication functions on surface ships, submarines, fixed and mobile land sites, and aircraft. The software for the system represents a large initial investment and will require a large investment in maintenance over the system life. Two companies were awarded contracts to build competing prototypes.

2. Technology Adopted

Software engineers from NRL were asked by the Navy project manager to consult on the software development. This was done after the contracts had been awarded to the two companies, after both companies had already decided on their software design and coding practices, and after a third company had already been contracted to provide software engineering advice to the project manager.

NRL found that both companies' developments had the typical set of software problems. Requirements were being written by one group at the same time that the software development was being done independently by another group. The requirements were being defined in terms of implementations making it difficult, if not impossible, for maintainers to distinguish real requirements from mere implementation decisions. Facts describing the interfaces with external systems (the satellite, networks, modems, etc.) were not isolated from each other or from the behavioral requirements, so that changes in the external systems and protocols could affect most of the software modules. Software design reviews for the military project managers were held well after many crucial decisions were embedded in the system design and their correction would have added months to the schedule.

3. <u>How Technology Was Transferred</u>

Although NRL personnel worked with the contractors and the Navy project managers for many months, there was no direct technology transfer.

4. <u>Obstacles to Transfer</u>

The major obstacle to technology transfer was that the Navy had committed insufficient resources to the management of the project. While the total being spent on software development by the two contractors was in the range of \$5M-\$10M, the Navy had only one engineer monitoring the software development, in addition to his other responsibilities on the project. This resulted in a situation in which those who were advocating the use of a strict software engineering discipline were not in a position to enforce its use.

The Navy gave each of the contractors the resources to field a management team that had a considerable edge in debating issues with the Navy's management team. As in all such cases, even though the Government and the contractors had many identical goals, there were instances where some goals were different. Post-deployment maintenance is a good example. It is to the contractor's advantage to design and implement the system so that he has the only people capable of maintaining the fielded system. He has no incentive for making the system easy to maintain by the Government or by another contractor.

The language in the current military standards tends to mandate form and organization rather than contents and quality. Further, in many instances, MIL-STD 1679 encourages presenting the requirements in terms of implementation.

The software engineering team was not in place before the contract was awarded. Work on the system design had already begun without any interactions between the software engineering team and the developer. Thus, the developer could claim that the disciplines advocated were not a part of his contract.

The contractor who was already providing advice to the project manager when NRL's aid was solicited viewed the NRL personnel as competition. Thus, the project manager, who was already overloaded trying to resolve conflicts over technical software issues, found himself arbitrating a three-way debate over software engineering issues.

The schedule for software design reviews were such that many issues were already decided by the contractor before the review. This resulted in claims that changes dictated by the Government at the reviews would add costs to the project or cause delays.

Two concerns of any project manager are that the imposition of the software engineering controls will increase the development costs of the system and that the contractor may use the imposition of the controls as an excuse to increase the development costs to cover errors he has already made. While the project manager may have been convinced that the methodologies would reduce the life-cycle cost of the entire system, the only costs that he is held responsible for are the current development costs.

5. <u>Aids To Transfer</u>

None

6. <u>Results</u>

There were no direct benefits from the effort. There were some indirect benefits that resulted form NRL's participation in design reviews and interactions with the developers, Navy managers, and support personnel.

Conclusions

NRL's experience in transitioning the SCR technology has proven that the overwhelming factor in determining the success of a technology transfer effort is the willingness of the responsible management to try something new, and all project engineers are reluctant to accept new, unproven methodologies. Thus, a difficult but crucial task faced by software engineering practitioners is that of providing proof that the technology being developed is useful, practical, and cost-effective. This proof must be presented in a convincing, unquestionable form. The project engineer must be presented with demonstrations of working models -- not promises. Without convincing demonstrations and working models that he can emulate, he will not be willing to experiment with new concepts and there will be no technology transfer.

Given a willingness of management to listen to new ideas, the following are ways in which to enhance the chances of transfer by making the technology as attractive as possible:

a. **Quantitatively Specify the Benefits Offered by the Technology.** It is purely a management decision whether the cost of a new technology will outweigh its benefits, but enough information must be available so that management can make a reasonable decision. This must be quantitative data; anecdotes are not sufficient. For example, we claim that the SCR methodology reduces the effort required to make changes in software and reduces the maintenance cost. To prove this, the effort required for software changes during development is being measured. After the development is complete, the effort required for a large set of changes to the "SCR" A-7 software will be compared with the effort to make the same set of changes to the "old" A-7 software.

b. **Provide a High Quality Model.** Software methodology exists in a community of engineers, and engineers work from models. For example, no one builds a bridge or a house or an airplane from first principles; rather, a (usually small) set of new ideas is incorporated into features from the last bridge or house or airplane. The last instance serves as a model, transferring technology (that set of ideas that were new to it) to the new effort. So it is with software engineering technology transfer. It is not enough to explain the technology or to write academic papers that describe it. It must be exemplified in a model. More than half the projects we know of that have adopted the SCR methodology have done so without any contact with NRL at all; They have relied completely on the models published by the project -- the examples of requirements specification, information-hiding module interfaces, etc.

c. <u>Reduce the Apparent Risk of the Technology.</u> In addition to providing the technological models for engineers to emulate, the worked-out examples provide proof that the technology can work; this decreases the number of unknowns faced by management and, hence, the apparent risk. The apparent risk also decreases as the number of customers and experience with the technology increases. Managers have little incentive to try something new, because there is no penalty for failing to do better than one's colleagues. However, there is a penalty for failing in an untried fashion, and so the perceived risk must be made small. The acceptance and use of portions of the SCR technology by companies like Bell Labs, SofTech, Grumman, Bell Northern, Tektronix, and TRW have helped decrease the apparent risk of the total technology package.

d. <u>Provide an Expert Consultant.</u> This is a special case of reducing the perceived risk, because the manager knows that when questions arise someone will be there to answer them. An expert will increase the quality of the model as seen by the customer, but absolutely cannot supplant the model.

These techniques are in order of importance. The project manager must be convinced that the technology's benefits outweigh the risks and are worth the costs. If he is not, then the matter is closed. If he is convinced, then it is essential to provide a high quality model for emulation and to provide data and examples to reduce his perceived risk. Providing an expert consultant helps, but usually is not essential.

Compiler Technology Insertion Networks Study
Richard A. DeMillo
February 1984

This case study tracks the four technical developments that have lead to the current state of engineering practice for compiler technology (SOP): (1) mathematics, (2) programming languages, (3) compiler theory, (4) compiler engineering practice.

The technology insertion process shown here only follows the insertion of lexical analysis and parser generator technology from its mathematical roots to common practice. The flow of critical events is shown in Figure 1. The arrows in Figure 1 indicate events which influenced other developments. It is assumed that there are downward arrows along each vertical track.

A complete analysis of all of compiling would also treat portability and optimization, culminating in the first commercial production-quality compiler-compiler.

Technology History

1937 Mathematics: Turing (41) defines and proves the existence of "universal" machines; these turn out to be compilers/interpreters for abstract machines and languages.

1938

1939

1940

1941 Mathematics: Church (10) describes the "lambda calculus", a formal system of computation notation that forms the basis of the programming language LISP and has a profound influence on programming language semantics, particularly denotational semantics.

1942

1943 Mathematics: McCullough and Pitts (34) publish a description of their "neural net" model. Although the notation and terminology undergo massive changes, the theory of finite automata stems from this paper.

1944

1945

1946

1947 Mathematics: Post (37) defines "production systems", a direct predecessor of the Chomsky formalization of grammar.

1948

1949

1950 Mathematics: Markov (34) publishes a "theory of algorithms" based on a generalized notion of rewriting. Although similar in spirit to Post's system, Markov investigates the properties of his model more thoroughly. This work eventually forms the basis of the theory of formal grammars as well as the design of a number of programming languages (e.g., SNOBOL (21)).

1951

1952

1953

1954

1955

1956 Mathematics: Chomsky (8) defines the basic model of modern formal language theory and sets out the hierarchy of languages that includes the regular, context free and context sensitive languages. Kleene (27) describes the theory of regular events and invents the "star" notation.

1957 State-of-Practice: Backus (2) describes a very early Fortran compiler. This is a practical compiler that translates Fortran to IBM 704 machine language.

1958

1959 Mathematics: Rabin and Scott (38) publish their model of finite automata and outline the problems which will occupy researchers for the next decade. The theory of finite automata can be said to begin at this point.

Language Definition: Backus (3) publishes what is probably the first -- and certainly the most extensive -- application of formal language theory to the definition of a programming language. This notation is adapted by Nauer of the final Algol 60 report to the BNF notation which becomes the standard for language definition.

1960 State-of-Practice: Dijkstra (12) and Bauer and Samelson (5) describe the run-time stack implementation of Algol 60, solving -- among other things -- the problem of how to implement block structuring. Although Dijkstra has been given extensive credit for this idea, at least one member of Algol committee reports that the run time stack implementation for block structured constructs was invented by an unknown Siemens engineer who attended a crucial meeting of the committee, described the solution, and was never heard from again.

1961 State of Practice: E.T. Irons is the intellectual father of automated compiler production. His classic paper (24) marks the first coherent description of syntax directed compilation and also contains a description of such a compiler for Algol 60.

1962 Mathematics: Brooker and Morris (6) publish the first general solution to the general context free language parsing problem. Chomsky (9) proves that pushdown automation recognition characterizes the context free languages. This is a theoretical demonstration that stack oriented translation systems are adequate. Kuno and Oettinger (30) outline the concept of top down parsing. Although (30) deals with the parsing of natural languages, the notion of recursive descent parsing for computer languages can be traced to this paper.

State-of-Practice: Paul (36) proposes a processing mechanism for Algol compilers that brings the efficient compilation of Algol within the realizable state of practice. The particular technique described is the direct, forerunner of bounded context parsing.

1963 Language Definition: Under the editorship of Nauer, the revised report on Algol appears in print. This is the most complete attempt to date to completely and unambiguously define the syntax and semantics of a programming language. In spite of many shortcomings and errors, this report and the notation contained therein is the standard by which all other definitions are judged until the early 1970's.

Theory of Compiling: Floyd's paper (16) formalizes the concept of operator precedence and places it in the context of formal language theory. Prior to this paper operator precedence had been used intuitively in compiler design, but attempts to justify its use were usually muddled and incorrect. In addition, Eickel, Paul, Bauer and Samuelson (15), publish an explicit definition of bounded context parsing algorithms. It is also explicit in this paper that automated parser production is an achievable goal.

State-of-Practice: Brooker and Morris (7) announce the first system that produces parsers automatically. Their system uses recursive descent parsing.

1964 Mathematics: Domolki (13) derives the first practical algorithm for grammar-driven parsing of arbitrary context free languages.

Language Definition: Floyd's survey (17) of the application of formal techniques to language definition and compiler design appears. Since there are virtually no textbooks in this area, Floyd's paper is the main vehicle for disseminating the rapidly improving state-of-the-art to the engineering community. Many of the algorithms discussed by Floyd are inserted into widespread practice on the strength of this exposition.

Theory of Compiling: Floyd (18) publishes the most general, correct definition of bounded context grammars.

State-of-Practice: Randell and Russell publish a complete description of an Algol 60 compiler. It is possible at this point for a competent programmer to implement a working Algol compiler using only the technology described.

1965 Mathematics: Earley announces his optimal solution to the general context free parsing problem (14). This table driven algorithm is suitable for inclusion in compilers and is used in one of the first Ada compilers. In a paper that generalizes all previous attempts to characterize situations in which languages can be determininstically parsed bottom up, Knuth (28) defines the class of LR(k) grammars: those that can be deterministically parsed bottom-up, left-to-right, using at most k symbols lookahead. Hereafter, all serious bottom-up parsers concentrate on LR(k) parsing for small values of k.

1966 Mathematics: Ginsbert (20) publishes the first textbook to systematically treat the theory of context free languages. This marks the beginning of an explosive interest in the subject and its insertion into standard graduate curricula.

Theory of Compiling: Wirth and Weber publish a definition of the Algol-like language EULER (42). Part 1 of this paper is notable for its introduction of "simple" precedence relations, a concept which significantly reduces the amount of effort to create and execute parsers. Variations of simple precedence parsing develop very rapidly and are put to immediate use.

1967 Language Definition: Galler and Perlis (19) invent a notation for syntactic and semantic extensibility for Algol-like languages. Although the notation is subsequently discarded as too cumbersome, such basic notions as encapsulation and separate compilation owe their roots to this paper. Standish's thesis (40) integrates many concepts which are vaguely articulated in (19) into a set of coherent design principles for extensibility and data definition. Modern semantic treatments of types appear here in a primitive form.

1968 Mathematics: Lewis and Stearns (32) define the class of LL(k) grammars, providing the top-down analog of the LR(k) parsing. Hereafter, virtually all recursive descent parsing algorithms concentrate on the "no backup" case, LL(1). Knuth (29) defines a scheme for attaching sematics to context free productions. This is the first use of "attribute" grammars and allows the automation of semantics production during compilation.

State-of-Practice: The structure of the algol W compiler is published (4). Of particular significance is the appearance of simple precedence parsing in a "production" quality compiler. The automatic production of lexical analyzers is brought within the state of practice with the announcement by Johnson, Porter, Ackley, and Ross (26) of the development of AED RWORD, a system that accepts regular expressions and produces finite-state acceptors. This system is used in several compilers as the lexical analyzer.

1969 Mathematics: Hopcroft and Ullman (23) publish their "Theory of Formal Languages and their Relation to Automata" which for over a decade is the standard text and reference in formal language theory.

Language Definition: In the ten years since the appearance of the first formal syntactic definition of a programming language, there has been little progress in semantic definition. In (32), Lucas and Walk apply an operational semantics developed by IBM Research in Vienna to the definition of the Programming Language PL/I.

1970 Mathematics: Scott (39) describes a method for giving a consistent semantics to programming language constructs. This is the starting point for the "denotational semantics" of programming languages.

1971 Mathematics: DeRemer (11) defines the "simple" LR(k) grammars. Most bottom up parsing schemes are subsequently oriented to this special subcase of the LR(k) grammars. Parsers for LR(k) grammars are especially suitable for automated production.

1972 Mathematics: Heck, and Ullman (22) publish the basis of flow graph reducilibility theory. This paper form is the foundation for virtually all efficient optimization schemes that used reducibility as the primary operation.

1973 Theory of Compiling: With the appearance of the two volume textbook of Aho and Ullman (1), the broad outlines of the theory of compiling are understood and can be communicated to any graduate student in software engineering.

1974

1975 State-of-Practice: Lesk describes the LEX tool (31). LEX is a regular expression based lexical analyzer generator that is included in the standard release of Berkely Unix(tm). Almost simultaneously, Johnson (25) announces the inclusion of an LR(1) bottom-up parser generator called YACC in Unix. With a commercial distribution of an automated lexical analyzer and parser generator, this portion of compiler production is completely automated.

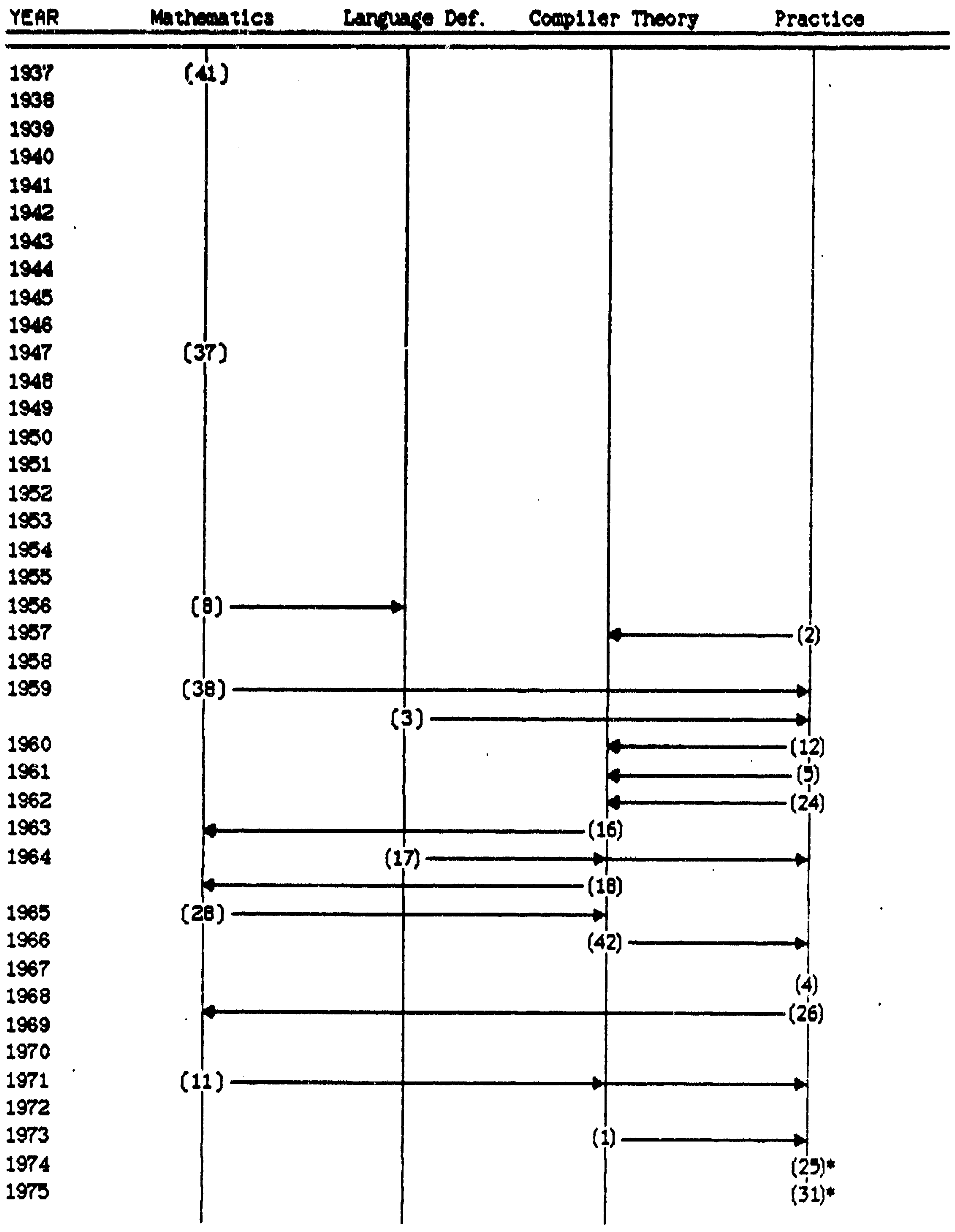


Figure 1. The Insertion Network for Commercial Compiler-Compiler (*)

References

(1) A. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling, (2 Volumes)," 1972-3, Prentice-Hall, Englewood Cliffs, New Jersey.

(2) J.W. Backus, et. al., "The Fortran Automatic Coding System", Proceedings of the Western Joint Computer Conference, 1957, 188-198.

(3) J.W. Backus, "The Syntax and Semantics of the Proposed Algebraic Language of the Zurich ACM-GAMM Conference," Proceedings of an International Conference on Information Processsing, UNESCO, 1959, 125-132.

(4) H.Bauer, S. Becker, and S. Graham, "ALGOL W Implementation", 1968, Technical Report, CS-90, Stanford University, Department of Computer Science, Stanford, California.

(5) H. Bauer and K. Samelson, "Sequential Formula Translation," communications of the ACM, 1960, v.2, 76-83.

(6) R. Brooker and D. Morris, "A General Translation Program for Pharase Structure Languages," Journal of the ACM, 1962, v.9, 1-10.

(7) R. Brooker and D. Morris, "The Compiler-Compiler," Annual Review in Automatic Programming, 1963, v. 3, 229-275.

(8) N. Chomsky "Three Models for the Description of Language," IEEE Transactions on Information Theory, 1956, V.2, 113-124.

(9) N. Chomksy "Context Free Grammars and Pushdown Storage," Quarterly Progress Report Number 65, 1962, Research Laboratory of Electronics, Massachussets Institute of Technology, Cambridge, Massachusetts.

(10) A. Church, "The Caluli of Lambda Conversion," Annual of Mathematics Studies, v.6, 1941, Princeton University Press, Princceton, N.J.

(11) F.L. DeRemar, "Simple LR(k) Grammars," Communications of the ACM, 1971, v. 14, 453-460.

(12) E.W. Dijkstra, "Recursive Programming," Numerische Mathematik, 1960, v.2, 312-318.

(13) B.Domolki, "An Algorithm for Syntactic Analysis," Computational Linguist, 1964, v.3, 29-46.

(14) J. Earley, "Generating a Recognizer for a BNF Grammar," Carnegie Institute of Technology, Technical Report, 1965, Pittsburgh, Pennsylvania, June, 1965.

(15) J. Eickel, M. Paul, F. Bauer, and K. Samuelson, "A Syntax Controlled Generator of Formal Language Processors," Communications of the ACM, 1963, v.6, 451-455.

(16) R.W. Floyd, "Syntactic Analysis and Operator Precedence," Journal of the ACM, 1963, v.10, 316-333.

(17) R. Floyd, "The Synax of Programming languages -- A Survey," IEEE Transactions on electronic Computers, 1964, v.EC-13, 346-353.

(18) R. Floyd, "Bounded Context Syntax Analysis," Communications of the ACM, 1964, v. 7, 62-67.

(19) B.A. Galler and A.J. Perlis, "A Proposal for Definitions in ALGOL," Communications of the ACM, 1967, v.10, 204-219.

(20) S. Ginsberg, "The Mathematical Theory of Context Free Languages," McGraw-Hill Book Company, 1966, New York.

(21) R.E. Griswold, J.F. Poage, and I.P. Polanski, "The SNOBOL 4 Programming Language," Second Edition, 1971, Prentice-Hall, Englewood Cliffs, N.J.

(22) M.S. Hecht, and J.D. Ullman, "Flow Graph Reducibility", Siam Journal of Computing, 1972, v.1, 199-202.

(23) J. Hopcroft and J.D. Ullman, "Formal Languages and Their Relation to Automata," 1969, Addison-Wesley, New York.

(24) E.T. Irons "A Syntax Directed Compiler for Algol 60," Communications of the ACM, 1960, v.4, 51-55.

(25) S.C. Johnson, "YACC - Yet Another Compiler Compiler", computer Science Technical Report 32, 1975, Bell Telephone Laboratories, Murray Hill, New Jersey

(26) W.L. Johnson, J.H. Porter, S.I. Ackley, and D.T. Ross, "Generation of Efficient, Lexical Processors Using Finite State Automatic Techniques," Communicatons of the ACM, 1968, v.11, 805-813.

(27) S.C. Kleene, "Representation of Events in Nerve Nets," Automata Studies (edited by C. Shannon and J. McCarthy), 1956, Princeton University Press, Princeton, N.J., 3-40.

(28) D.E. Knuth, "On the Translation of Languages from Left to Right", Information and Control, 1965, v.8, 607-639.

(29) D.E. Knuth, "Sematics of Context-Free Languages," Mathematical System Theory, 1968, v.2, 127-146.

(30) S. Kuno and A. Oettinger, "Multiple-Path Syntactic Analyzer," Information Processing 62, 1962, North-Holland, Amsterdam, Netherlands, 306-311.

(31) M. Lesk, "LEX-A Lexical Analyser Generator", computer Science Technical Report 31, 1975, Bell Telephone laboratories, Murray Hill, N.J.

(32) P.M. Lewis and R. E. Stearns, "Syntax-Directed Transduction," Journal of the ACM, 1968, v.15, 465-488.

(33) P. Lucas and K. Walk, "On the Formal Description of PL/I, "Annual Review in Automatic Programming," 1969, v. 6, 105-182.

(34) A.A. Markov, "The Theory of Algorithms," Proceedings of the Steklov Institute, 1951, v.38, 176-189, in Russian.

(35) W.S. McCullough and E. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," Bulletin of Mathematical Biophysics, 1943, v.5, 115-133.

(36) M. Paul, "ALGOL 60 Processors and a Processor Generator," IFIP Congress, 1962, Munich, 493-497.

(37) E.L. Post "Recursive Unsolvability of a Problem of Thue," Journal of Symbolic Logic, 1947, v.12, 1-11.

(38) M.O. Rabin and D.S. Scott, "Finite Automata and Their Decision Problems", IBM Journal of Research and Development, 1959, v.3, pp. 114-125.

(39) D.S. Scott, "Outline of a Mathematical Theory of Computation," Proceedings of the 4th Princeton conference on Information Science and Systems, 1970.

(40) T.A. Standish, "A Data Definition Facility for Programming Languages," Ph.D. Thesis, 1967. Computer Science Department, Carngie-Mellon University, Pittsburt, P.A.

(41) A.M. Turing, "On Computable Numbers with an Application to the Entscheidungsproblem," Proceedings of the London Mathematical Society, 1937, ser.2, v.42, 230-265.

(42) N. Wirth and H. Weber, "EULER: A Generalization of Algol, and its Formal Definition (Part 1)," Communications of the ACM, 1966, v. 9, 13-25.

TECHNOLOGY CASE STUDY
SOFTWARE ENGINEERING CONCEPTS

Dr. John H. Manley

Computing Technology Transition, Inc.

82 Concord Drive
Madison, Connecticut 06443
(203) 421-4585

MANAGEMENT OVERVIEW

The fundamental purpose of software engineering is to put order and discipline into the process of planning, developing and supporting software for use in industrial, military, Government and consumer products. Since every sector of modern society is now dependent on software in one form or another, software products must be virtually error-free, yet be produced as economically as possible. The time has therefore arrived, that software engineering is no longer a luxury, but a necessity for survival...especially with respect to United States mission critical systems.

The concept of "software engineering" was first documented at a 1968 NATO-sponsored conference by a German professor, Fritz Bauer. His simple but yet powerful statement was that software engineering is:

"The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines."

This concept has been embellished since that time by the professional community, but has not been changed. Some of the basic "engineering principles" that have become generally accepted as being appropriate to software engineering are as follows:

a. A SYSTEM PERSPECTIVE is taken with respect to every task and activity pertaining to software planning, development and support. This requires the use of a LIFE CYCLE FRAMEWORK so that every operation can be viewed in perspective on how it impacts the overall software system development and support operation, especially with relation to every major software project's total cost and schedule.

This report was prepared as part of work performed for the Department of Defense STARS Joint Program Office by the Computer and Software Engineering Division of the Institute for Defense Analyses.

b. Methods and tocls are used to SIMPLIFY COMPLEXITY (both management and technical). They must be based on provable principles that can generate reproducible results. This is the reason that STRUCTURED METHODS such as work breakdown structures and structured programming are very important parts of any software engineering tool kit.

c. MANAGEMENT DISCIPLINE is instilled through the use of STANDARD PRACTICES and MEASUREMENT SYSTEMS to help improve cost and schedule estimates and better control interacting life cycle processes.

d. A constant focus on PRODUCTIVITY IMPROVEMENT and QUALITY CONTROL help keep management alert to any new method, tool or technique that may offer improvement.

e. AUTOMATION and REUSE OF STANDARD PARTS (reusable code) are principles inherent in every engineering profession, including software engineering.

f. The newest concept to emerge with respect to software engineering is that of SOFTWARE "FACTORIES." This is an ideal industrial environment wherein new and complex defect-free software systems (not only user applications) can be "manufactured" orders of magnitude faster than is possible today.

In 1984, active users of software engineering (those that profess to practice at least some of the principles) include:

a. Most defense and space system builders.

b. Most leading commercial aerospace companies.

Users that are experimenting with software engineering are:

a. Many developers of commercial high risk systems such as those that control airline reservations, public telephone switching, electronic funds transfer, and power or natural gas distribution.

b. Some large scale data processing/management or business information system operations, such as exist in Department of Defense logistics organizations, and in the insurance and finance industries.

c. A small number of commercial engineered products companies (just now increasing their use of embedded microcomputers in products such as automobiles, major applicances and the like).

What is the reason for the expanding uses of software engineering? Simply stated it involves the benefits derived from the application of the above stated engineering principles to the software life cycle...a series of well-defined steps that comprise the essential elements of software planning, development and support.

Since the benefits that can accrue to those who choose to practice software engineering are so extensive, the challenge for the Department of Defense is to improve the current state of practice as rapidly as possible within the Defense community. Some conceptual approaches that have been advocated as being the most effective are as follows:

a. Develop a coherent model of software planning, development and support that can be applied successfully within a given organization.

b. Identify, collect and integrate sets of life cycle support methodologies into a consistent overall software engineering process that is compatible with an organization's existing "standard" approach to planning, developing and supporting software on a professional basis. This defines and codifies a tailored standard life cycle approach or SOFTWARE LIFE CYCLE FRAMEWORK.

c. Connect (by first coordinating and then integrating) individual organization tools and methods into the life cycle framework by defining subset methodologies that appear useful to codify. Some SOFTWARE LIFE CYCLE SUPPORT METHODOLOGIES that have been found to be highly useful include requirements tracing and validation (from user requirements to operational system); software configuration control; test planning, execution and results analysis; and integrated data base management systems to include design dictionary and data dictionary functions.

d. Select and then automate required paper-intensive manual methodologies such as those used in planning, software design and project management. Insure that during the automation step that each of the INDIVIDUAL AUTOMATED METHODS is integrated into the life cycle framework.

e. Select and automate (as necessary) STAND-ALONE TOOLS that are compatible with the overall life cycle framework and selected life cycle support methodologies.

f. As an operational imperative, develop a consistent and well integrated set of COMPUTER AIDED SOFTWARE ENGINEERING processes, methodologies, individual methods and stand-alone tools. The undesirable alternative is an eclectic and burdensome mixture of automated and manual methods.

g. Throughout the above process, educate and train the software planning, development and support organizations, from executives to bench-level workers to effectively institute the "cultural change" that will result from instituting the SOFTWARE ENGINEERING ENVIRONMENT that will evolve during the above implementation activity.

Although software engineering is a relatively new concept, applications of software engineering principles are becoming increasingly accepted and used throughout the computing industry. However, some implementation problems are evident, primarily in the area of large scale method and tool integration.

Problems associated with software engineering implementation were addressed in 1983 by STARS-sponsored working groups such as the Rights In Data Working Group and the Software Engineering Institute Working Group. Also in 1983,the IEEE Computer Society studied the problem in an internationally-attended workshop. Concepts for solution are now emerging in the form of automation aids such as "computer aided software engineering," enhanced human engineering of software engineering environments and tools, and organizational solutions such as the proposed Software Engineering Institute.

As more effective computer aids for software engineering are developed over time, most of the present technical implementation problems will be overcome. This will permit movement toward the next major conceptual software engineering milestone, the development and implementation of software "factories."

HISTORY OF SOFTWARE ENGINEERING CONCEPTS
(Time Line)

1940's	Military computing needs for ballistic tables, astronomic navigation tables, and so forth satisfied by newly invented digital machines. Software very primitive and simple. "Bug" was a real moth!
1947	First public symposium on computers sponsored by Navy Bureau of Ordnance (word computer not yet used -- "large scale digital calculating machinery").
1950's	Focus on automation of existing manual business systems through "data processing." Functional programming also taking place in the scientific and military communities. Much "software" was electromechanical.
1960's	In projects like SAGE, large assemblages of programs for military systems were being attempted, rather than developing single programs or small sets of programs. Lessons learned generated the need for a new philosophical viewpoint which would enable software designers to create systems (not just single programs) that worked.
1965	(Brooks Bill) P.L. 89-306 codifies Government computer hardware procurement to reduce duplication and cost to Government. Creates eventual problems with "embedded computer" developments as underlying concepts apply to data processing community and not an engineering approach to hardware/software systems.
Late 1960's	Concepts that were "discovered"...difference between logical design (i.e., abstract, conceptual) and physical design (i.e., blueprint, one-for-one correspondence to what will be built), philosophies of design (e.g., top-down) and the use of prototypes from which to learn and protect an investment.
1968	"Software engineering" coined by Fritz Bauer at a NATO Conference in Garmisch, West Germany. Essence is to use "sound engineering principles" to develop software.
1968	"Quantitative measurement" of software quality proposed at ACM National Conference.
1969	Collected papers by Julius Tou on software engineering issues "in the small" [i.e., technologies] no major emphasis on management or systems engineering...recognized problem however.

1969-70	Air Force Systems Command mission analysis CCIP-85 delineates in a nine-volume report the need for an engineering approach to command and control system software development and support.
1970-72	Lessons learned at IBM promoted concepts in the form of top-down design, chief programmer teams, structured programming [,structured design, design walk through, software design language, unit development folders, quality assurance/configuration management and life-cycle maintenance.
1973	"Embedded computer system" coined by John Manley to separate concepts of software development as integral part of an engineered product as opposed to data processing oriented applications programs.
1973	Symposium on "High Cost of Software" at Monterey Naval Postgraduate School determined that management of software for military systems a major problem that needs attention.
1973	Air Force Systems Command established special office to oversee software planning for "embedded computer systems."
1973	First set of collected readings on software engineering edited by Fritz Bauer...used as early advanced course text.
1973	Air Force Systems Command proposed an Air Force Regulation "800-xx" to codify embedded computer system software development as a part of a systems engineering process. Separates acquisition from data processing oriented "Brooks Bill."
1973	Havatny and Janos stated the "real breakthrough will be when `art' becomes `engineering practice,' i.e., when the process of planning and implementing a new set of programs comprising a new software system is as much a routine engineering activity as it is with any other product."
1974	U.S. Air Force published policy guidance on engineering approach to embedded computer system software.
1975	U.S. Air Force published AFR 800-14...codified "800.xx"
1975	First International Conference on Software Engineering held in Washington, D.C. sponsored by the IEEE Computer Society Technical Committee on Software Engineering.

1975	Joint Logistics Commanders Software Reliability Work Group (SRWG) Report recommends detailed plan to OSD to extend AFR 800-14 and Navy MIL STD 1679 embedded computer system software development concepts to Department of Defense level.
1975	Office of Secretary of Defense (DDR&E) proposed a single programming language for embedded computer system software. Dubbed "DoD-1."
1976	ACM SIGSOFT (Special Interest Group on Software Engineering) established. First issue of Software Engineering Notes published.
1976	DoDD 5000.29 published as result of JLC SRWG proposal and testing in form of a five-year software improvement plan ("Blue Book"). Separates "embedded computer system" acquisition from Brooks Bill (1965).
1976	Set of allowable programming languages limited by the Department of Defense through publication of DoDI 5000.31.
1977	NASA Software Engineering Laboratory established.
1979	First formal textbooks on "software engineering" available to educators (e.g., Jensen and Tonies, Zelkowitz et.al.).
Late 1970's	Concept of the need to solve "real world" problems. This resulted in two general approaches to software engineering... Methods (strategies, recommendations, or guidelines based on a philosophical view) and techniques (tactics or well-advised "tricks of the trade").
Late 1970's	IEEE Computer Society working to define and get certification for a standard undergraduate degree program in Software Engineering. Concept that software can be "engineered" meeting heavy resistance from "real" engineers.
Late 1970's	Initial university curricula (sets of courses) on software engineering begin trial use.
1980	Wang Institute masters degree program in Software Engineering certified by Commonwealth of Massachusetts.
1980	Harlan Mills distinguishes software engineering from computer science. Advocates separation in universities.

1981	"Warner Amendment" to P.L. 96-511 completes the separation of embedded computer system acquisition from Brooks Act. "Embedded computer systems" become a subset of an enlarged set of "mission critical systems" [i.e., Intelligence, Cryptologic, Command and Control, Integral Part of Weapons System (original "embedded" concept), Critical to Direct Fulfillment of Military or Intelligence Missions]. These systems are to be "system engineered."
1981	First software economics textbook by Barry Boehm.
1982	First commercially-oriented software engineering textbook by Roger Pressman...focused primarily on commercial engineered products software development.
1982	Department of Defense begins discussions on desirability of establishing a Software Engineering Institute.
1983	Beginning of the popularization of software engineering. "Software engineering" commonly appears in the trade press, technical literature and in professional conversation.
1983	Ada directed by Office of Secretary of Defense to be the single programming language for "mission critical" systems.
1984	"Computer aided software engineering" (CASE) defined by John Manley in an award-winning paper at a Federal data processing conference in Washington, D.C. Concept is the need for tools (computer aids) to effectively implement software engineering in production environments.
1984	A software "factory" approach for Department of Defense proposed by Edith Martin at a conference in Washington, D.C. Emphasis on building on a software engineering base to be able to reuse code and fully implement computer aided software engineering (CASE).
1984	Department of Defense formally announces its intention to establish a Software Engineering Institute through competitive procurement.

TECHNICAL APPENDIX

****

The following technical back-up data is provided to help justify the summary statements included in the case study Management Overview.

****

This Appendix contains selected quotations and other excerpts from the professional literature written by a great number of individuals, most of whom are considered to be software engineering pioneers and principal spokespersons.

Since the field is so new (less than 20 years old), no single person or group has yet emerged that can be quoted as _the_ authority on software engineering. Therefore, it is hoped that the following material will provide at least a general understanding of what the "concept of software engineering" is all about.

The Appendix is organized into two overall sections as follows:

1. Definitions of Software Engineering.

This section is intended to show how the original statement by Bauer in 1968 has not been substantially changed over time.

2. Software Engineering Concepts.

This section contains excerpts from various professionals which collectively provides insight into the problems software engineering is supposed to solve, together with some recommended solutions.

DEFINITIONS OF SOFTWARE ENGINEERING

Bauer, F. L. (1972) -- ...The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

Jensen, R. W. and Tonies, C. C. (1-1979) -- The software engineer must be able to determine the actual needs of a user; select a general approach to the system development; analyze requirements to determine and resolve conflicts; establish a design to achieve the desired performance within constraints imposed by cost, schedule, and operating environment; develop new technical solutions; and, finally, manage a group of individuals with a wide range of personalities, disciplines, and goals.

Mills, H. D. (1-1980) -- ...software engineering requires both software and engineering as essential components. By software we mean not only computer programs, but all other related documentation including user procedures, requirements, specifications, and software design. And by engineering, we mean a body of knowledge and discipline comparable to other engineering curricula at universities today, for example, electrical engineering or chemical engineering.

We distinguish software engineering from computer science by the different goals of engineering and science in any field - practical construction and discovery. We distinguish software engineering from computer programming by a presence or not of engineering-level discipline. Software engineering is based on computer science and computer programming, but is different from either of them.

Lehman, M. M. (1-1980) -- ...as mankind relies more and more on the software that controls the computers that in turn guide society, it becomes crucial that people control absolutely the programs and the processes by which they are produced, throughout the useful life of the program. To achieve this requires insight, theory, models, methodologies, techniques, tools: a discipline. That is what software engineering is all about.

Peters, L. J. (1-1980) -- Software design is a branch of software engineering.

Belady. L. (1-1980) -- ...neither is there an agreed upon definition of software or of its engineering...[on the other hand]...all seem to accept the great variety of presently professed and applied approaches to improve software quality and to reduce cost of its development, maintenance and operation.

Freeman, H. and Lewis II, P. M. (2-1980) -- Software engineering -- the art, science, and discipline of producing reliable software efficiently...

Boehm, B. W. (1-1981) -- Our definition of software engineering is based on the definitions of software and engineering given in the current edition of "Webster's New Intercollegiate Dictionary [1979]:

o Software is the entire set of programs, procedures, and related documentation associated with a system and especially a computer system.

o Engineering is the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems, and processes.

Since the properties of matter and sources of energy over which software has control are embodied in the capabilities of computer equipment, we can combine the two definitions above as follows:

o Software engineering is the application of science and mathematics by which the capabilities of computer equipment is made useful to man via computer programs, procedures, and associated documentation.

Pressman, R. S. (1-1982) -- The software implementation of a problem solution...can be approached by using a set of techniques that are applicaton-independent. These techniques form the basis of a software engineering methodology...Software engineering is modeled on the time-proven techniques, methods, and controls associated with hardware development. Although fundamental differences do exist between hardware and software, the concepts associated with planning, development, review, and management control are similar for both system elements. The key objectives of software engineering are (1) a well-defined methodology that addresses a software life cycle of planning, development, and maintenance, (2) an established set of software components that documents each step in the life cycle and shows traceability from step to step, and (3) a set of predictable milestones that can be reviewed at regular intervals throughout the software life cycle.

SOFTWARE ENGINEERING CONCEPTS

Bauer, F. L. (2-1980) -- The term "software engineering" is now 12 years old [as of conference in 1979]; "computer science" as a new scientific discipline is perhaps 20 years old; both are based on the development of the modern computer which is not more than 40 years old.

In 1947, a symposium on "Large-Scale Digital Calculating Machinery" (the word computer was not yet used) was held at Harvard, organized by Professor Howard Aiken and sponsored by the Navy Department's Bureau of Ordnance; it was probably the first public symposium on computers...That is how computers started, with applications to firing tables, neutron diffusion, telecommunication, cryptography, and buzz-bombs - or should I say with Aiken, von Neumann, Stibitz, Turing, and Zuse.

[as of 1979] It has taken us a long time to arrive at our present state and that it will be a long time before we shall have genuine software engineering. My guess is that this will not be fully accomplished in the next 10 years [1989]. It needs a "change of culture." What I expect for the next 10 years, however, is more work on formalization and that its importance will be better understood...Indeed there is no way out: a chemist cannot work today without a quantum mechanics background, a classical engineer has to learn some mathematics, and a future software engineer will have to learn what formaliztion is and how to work with it.

Browne, J.C. (1-1980) -- ...software engineering is just now beginning to turn its attention to the problem of designing and implementing programs which meet performance goals and performance criteria. [SIGMETRICS, ACM Special Interest Group on Performance Measurement and Modeling was born from 1971 Conference on Performance Measurement and Modeling (Boston) sponsored by SIGOPS, ACM Special Interest Group on Operating Systems.

Hatvany, J. and Janos, J. (1-1980) -- ...real breakthrough will be when "art" becomes "engineering practice," i.e., when the process of planning and implementing a new set of programs comprising a new software system is as much a routine engineering activity as it is with any other product. [originally stated by the author in 1973].

Yeh, R. T. and Zave, P. (1-1980) -- ...we are optimistic about progress toward a disciplined approach to the requirements phase, although much more research effort, particularly in an experimental direction, is needed to make it commercially practical.

Mills, H. D. (1-1980) -- The full discipline of software engineering is not economically viable in every situation. Writing high-level programs in large well structured application systems is such an example. Such programming may well benefit from software engineering principles, but its challenges are more administrative than technical, more in the subject matter than in the software.

However, when a software package can be written for fifty thousand dollars, but costs five million to fix a single error because of a necessary recall of a dangerous consumer product, the product may well require a serious software engineering job, rather than a simple programming job of unpredictable quality.

Since the content of software is essentially logical, the foundations of software engineering are primarily mathematical - not the continuum mathematics underlying physics or chemistry, of course, but finite mathematics more discrete and algebraic than analytic in character...software engineering uses continuum mathematics only for convenient approximation, e.g., in probability or optimization theory.

The primary difficulty in software engineering is logical complexity. And the primary technique for dealing with complexity is structure. Because of the sheer volume of work to be done, software development requires two kinds of structuring, algebraic and organizational...the result of proper structuring is intellectual control, namely the ability to maintain perspective while dealing with detail, and to zoom in and out in software analysis and design.

The management of software engineering is primarily the management of a design process, and represents a most difficult intellectual activity. Even though the process is highly creative, it must be estimated and scheduled so that various parts of the design activities can be coordinated and integrated into a harmonious result, and so that users can plan on results as well.

In software engineering, there are two parts to an estimate - making a good estimate and making the estimate good. It is up to the software engineering manager to see that both parts are right, along with the right function and performance.

Musa, J. D. (1-1980) -- ...the field of software reliability metrics has made substantial progress in the last decade. It cannot yet provide a standard cookbook approach for widespread application...however it is clearly beyond the pure theory stage and it can provide practical dividends for those who make the modest investment in time required to learn and apply it.

Curtis, B. (1-1980) -- Software wizardry becomes an engineering discipline when scientific methods are applied to its development. The first step in applying these methods is modeling the important constructs and processes. When these constructs have been identified, the second step is to develop measurement techniques so that the language of mathematics can describe relationships among them. The testing of cause-effect relationships in a theoretical model requires the performance of critical experiments to eliminate alternative explanations of the phenomena. Even when possessed of supportive experimental evidence, our sermonizing should be cautious until we have established limits for the generalizability of our data.

There is no substitute for sound experimental evidence in arguing the benefits of a particular software engineering practice or in comparing the relative merits of several practices...scientific study of software engineering is young, and its rate of progress will improve as measurement techniques and experimental methods mature.

Distaso, J. R. (1-1980) -- Obtaining satisfactory software requirements remains the single largest obstacle to software project success...The key elements of this problem are the following:

1) continually changing user needs

2) scheduling difficulties of major system developments

3) communication barriers among users, system designers and software designers

4) lack of use of new generalized methodologies

5) misapplication of simulation

In early 1970's...lessons learned at IBM...presented methodology which included the concepts of top-down design, chief programmer teams, structured programming [,structured design, design walk through, software design language, unit development folders, quality assurance/configuration management, life-cycle maintenance]...These concepts received widespread acclaim in the literature, but did not as readily become accepted practices within the industry. Although a 1977 study by Holton implied a lack of widespread usage of these approaches at that time, Munson demonstrated at a 1979 workshop on project management that at least most aerospace and defense system contractors now use some version of these practices as a matter of policy.

As the 1970's were completed...many large projects are being completed on schedule and within cost; a methodology for controlling large developments is being employed in many places; and reliable cost models are being demonstrated and validated...it is realistically possible today to estimate, develop and field a large operational software system with a high probability of success...given reasonable development conditions, successful developments of monolithic systems should start becoming the norm. Unfortunately, the processing demands of the 1980's will not likely allow this utopian condition to last for long. Some of the areas which software managers should watch for in the 1980's include:

Requirements Definition...changing customer specifications

Distributed Processing...many monolithic system procedures and techniques not transferable or upgradable...multiplication of possible paths, intraprocess timing and deadlock, synchronization of data bases, nondeterminism of the programming problem...adds new complexity

Microprocessing...lessons of 1960's of need for higher level languages, structured developments, integration principles, etc., are being relearned

Personnel...growth of data-processing industry accelerated ...availability of trained personnel almost completely exceeded.

Super Systems...(FET, communication, command and control, etc.)...these systems, by their very nature, are beyond the scope of understanding of any person or team of persons...yet data-processing industry must demonstrate a capability to evolve this class of system to provide the services demanded by society while respecting the concerns of society for privacy and protection against "big government/business."

Government Regulations...solution of government to this perceived "problem" [software] is likely more regulation and standardization ...while not all bad, potential effect of overregulation at a time when innovation is required could be devastating.

Peters, L. (1-1980) --[Historical perspective]

Emphasis in early software development was on obtaining a program which worked. That is, it gave answers which agreed with accepted values or, where accepted values did not exist, saved large amounts of manual labor...1930's using comptometers and a lot of manual labor [1940's for astronomic and ballistic tables, 1950's for business data processing]

1960's...instead of developing single programs or small sets of programs, large assemblages of programs were being attempted. [SAGE for example] This ushered out the age of functional programming and ushered in the age of structure oriented programming...need for some philosophical viewpoint which would enable software designers to create systems (not just single programs) that worked; systems whose construction was aided by the design and not encumbered by it. ...concepts "discovered"...difference between logical design (i.e., abstract, conceptual) and physical design (i.e., blueprint, one-for-one correspondence to what will be built), philosophies of design (e.g., top-down) and the use of prototypes from which to learn and protect an investment. Much of this ferment peaked in the late 1960's and early 1970's with the advent of some specific methods and approaches to the problem of software design representations.

Still need to solve "real world" problems. This has resulted in yet another wave of approaches. This body of help now available [1980] falls into two classes-methods (strategies, recommendations, or guidelines based on a philosophical view) and techniques (tactics or well-advised "tricks of the trade").

SELECTED REFERENCES

1-1969: P. Naur and B. Randell, Eds., "Software Engineering: Report on a conference sponsored by the NATO Science Committee," (Garmisch, Germany), Oct 7-11, 1968. Brussels, Belgium: Scientific Affairs Division, NATO, 1969, 231 pp.

1-1979: R. W. Jensen and C. C. Tonies, Software Engineering, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

1-1980: Proceedings of the IEEE, Special Issue on Software Engineering, September 1980

2-1980: H. Freeman and Lewis II, P. M., Software Engineering, Academic Press, New York, 1980.

1-1981: B. Boehm, Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1981

1-1982: R. S. Pressman, Software Engineering - A Practitioner's Approach, McGraw-Hill Book Company, New York, 1982.

1-1983: Communications of the ACM, "Special 25th Anniversary Issue," Vol. 26, No. 1, Jan 1983.

1-1984: R. Fairley, Softwre Engineering, McGraw-Hill Book Company, New York, 1984 [in printing]

TECHNOLOGY CASE STUDY
SOFTWARE METRICS

Dr. John H. Manley

Computing Technology Transition, Inc.

82 Concord Drive
Madison, Connecticut 06443
(203) 421-4585

MANAGEMENT OVERVIEW

The primary function of software metrics is to assist management in planning and controlling medium to large scale software development projects. Since management measures such as headcount, labor cost, burden components, and so forth are always necessary, this report will only address special metrics required to plan and control software as a physical component of a system or as a product in its own right.

In order to plan any software development project, management wants answers to the following questions in quantitative terms:

1. How much software can we reuse? Although somewhat peripheral to the main topic, this is the most important question of all since it determines how much work can be avoided before commiting resources to new work.

2. How much new software must we produce?

3. How difficult will it be to produce?

4. How much will it cost to produce?

5. How long will it take to produce? Is this fast enough to meet our commitment, or meet an anticipated threat?

6. How difficult will it be to maintain once delivered to a customer? This is the second most important question for enlightened management to have answered up front since rapidly growing software inventories require increasing numbers of human and physical resources to maintain them. This burden must be analyzed carefully at the beginning of every new project to determine whether or not the projected increase will be manageable.

This report was prepared as part of work performed for the Department of Defense STARS Joint Program Office by the Computer and Software Engineering Division of the Institute for Defense Analyses.

In order to control a software development project once it is ongoing, we would like to know the following in quantitative terms:

1. How fast are we developing the software in each defined stage or phase of the process?

2. How good is the software product at each stage of the process?

3. How far have we come in developing the software?

4. How much farther do we have to go in developing the software?

Finally, management always wants to know if it can find means to improve any facet of operations in order to reduce costs. This includes simple "cost avoidance" and more complex "value engineering" activities commonly used in hardware engineering. The objective is to increase profits directly, or indirectly through lowering prices to improve competitive position and increase volumes with smaller margins.

Regardless of the tactic involved, all cost reduction activities require two major ingredients. First, a sound experimental methodology or approach to make valid comparisons between competing processes, methods, tools and techniques. Second, a valid baseline or starting criteria upon which proposed changes can be evaluated. Some of the questions pertaining to software product baselines that require quantitative answers are as follows:

1. What is our current software development productivity rate?

2. What is our current software product quality level?

The software metrics that have been proposed, experimented with in univerities and laboratories, or actually used in practice number in the thousands. To possibly oversimplify this very complex topical area, I will categorize software metrics into a relatively small set that can be related to answering the practical questions listed above. Therefore, the primary categories of software metrics are defined as follows:

A. Software Development Productivity (Actual)

   1. Physical Output per Unit Input

   2. Functional Output per Unit Input

B. Software Product Attributes (Actual)

   1. During Development

      a. Software Volume

b. Software Quality

c. Software Complexity

d. Software Testability

2. Upon Delivery to Customer/User

a. Software Volume

b. Software Quality

c. Software Complexity

d. Software Maintainability

3. During Operation (After Customer/User Acceptance)

a. Changes in Software Quality

b. Changes in Software Complexity

c. Changes in Software Maintainability

C. Predictive Software Metrics

1. Software Productivity Estimates

a. Physical Output per Unit Input

b. Functional Output per Unit Input

2. Software Quality Estimates (Examples)

a. Faults or Defects/Unit Volume at Delivery

b. Probability of Failure/Unit Time

c. Probability of Failure/Number of Transactions

d. Testability

e. Maintainability

f. Other "ilities"

3. Software Project Estimates

a. Cost to Complete (from any point in time)

b. Completion Date (from any point in time)

c. etc.

As can be inferred from this very abbreviated outline, an effective set of software metrics for any organization is not only difficult to initially define, but is even more difficult to develop and use without expert assistance.

Furthermore, since the management measures suggested above are most often (but not entirely) based on empirical statistics, a set of reliable software measures will take at least three to four years to establish in any organization. This assertion is primarily based on over 20 years of experience working with management measures to include setting up comprehensive software measurement systems both for Department of Defense organizations and large commercial firms. This assertion can also be verified by the literature (see the Technical Appendix to this Case Study).

Another major category of software metrics involves analytical approaches such as cost and schedule models, attempts at defining a software "science," software reliability models, complexity measures from a microscopic viewpoint, and so forth. Most of this work is still in an embryonic stage and, in my opinion, will progress much more slowly that the empirical efforts. For a better understanding of this area see the Technical Appendix to the Case Study.

In summary, software metrics are essential to establishing and improving software engineering and management practices for any software product. Since the state of practice is still quite primitive, considerable effort must be expended to put them into effective use.

HISTORY OF SOFTWARE METRICS
(Time Line)

1964	System Development Corporation recognizes importance of estimating cost of computer program production
1968	Quantitative measurement of computer program quality discussed at ACM National Conference
1969-70	U.S. Air Force command and control system Mission Analysis (CCIP-85) documents lack of quantitative knowledge about software -- stimulates research -- software metrics need established
1972	Halstead metrics concept first published -- first attempt at establishing a metrics based "software science" -- start of controversy
1972	TRW begins quantitative experiments on software reliability
1973	Software still touted as "invisible cloth" in the trade press (Datamation) -- considered unmeasurable in practice
circa 1974	IBM Santa Theresa Research Center begins collecting empirical data on commercial software productivity and quality
1975	Major Joint Logistics Commanders study of software reliability -- probabilistic definition of software reliability distinguished from hardware reliability metrics -- software metrics need reconfirmed
1976	First significant quantitative results of IBM commercial software quality improvement projects reported
1976	Software testing metrics concepts proposed by Mohanty and Adamowicz
1976	Software complexity measure proposed by McCabe
1977	NASA Software Engineering Laboratory established -- initial results reported
1977	Halstead's metrics refined -- in book form as "software science" -- still controversial
1977	IBM Federal Systems Division proposed measures for embedded computer system programming measurement and estimation

1978	Software data collection and analysis formalized at the Air Force Rome Air Development Center -- beginning of DACS (Data & Analysis Center for Software)
1978	IBM commercial software productivity and quality measurement successes reported
1980	ITT establishes a major software measurement program -- initial productivity and quality baseline established -- compared with DACS database (no other available)
1982	Halstead's "software science" still not accepted by software engineering community -- controversy continues
1982	User oriented software productivity metrics called "function points" established by IBM in commercial data processing environment
1983	Analysis of combined NASA/SEL and DACS productivity data initiated
1983	Software metrics confirmed as a major technical need by the Department of Defense STARS Program
1984	ITT's software productivity and quality results reported in the open literature for the first time -- very large data base established but still considered prototype system

At the present time, the commonly agreed state of software metrics is as follows:

1984	Complexity measures still being evaluated...no known commercial use
1984	"Function points" not understood by non-IBM users
1984	Experimental use of software productivity measures in commercial sector just beginning
1984	Almost no commercial use of software quality metrics
1984	Software metrics considered in the prototype stage of technical development -- productization still required before widespread use can be expected

TECHNICAL APPENDIX

The following technical back-up data is provided to help support the summary statements included in the case study Management Overview.

The Appendix is divided into metrics categories keyed to the summary statement. Only some of the areas are presented and appropriate references are contained at the end of the Appendix.

Even though this appendix is incomplete, it should provide enough depth to give the interested reader a feel for the complexity of the topical area and the need for further research. This will be especially apparent with respect to software quality.

As in any product, software included, "quality" connotes different things to different people. For example, a product developer's viewpoint is often substantially different from the user's. As a result, "quality metrics" are scattered throughout this Appendix, not by design, but of necessity.

Category A1

Software Development Productivity (Actual)
(Physical Output per Unit Input)

Software development productivity is often described in the trade literature in terms of the number of "lines of code" a programmer can produce per unit of time.

Since lines of code do not physically appear for counting until well into the software development life cycle, they are hardly representative of software productivity. But, as we will see later, this measure can be highly useful in another way.

The missing productivity metrics are those that measure physical outputs derived from carrying out activities such as requirements analysis, top level design, detailed design, integration, testing, configuration management, and any other tasks considered integral parts of the software development life cycle process.

This point has been recognized by many researchers including those conducting studies at the NASA Software Engineering Laboratory. For example, comparative research is reported in [BASILI-81] on physical outputs such as the following:

Documentation -- Measured in pages and is defined as the program design, test plans, user's guide, system description, and module descriptions.

Total Number of Modules -- Number of modules delivered in the final product. A module is further defined as a separately compilable entity, such as a subroutine, funtion, or BLOCK DATA unit.

Number of New Modules -- Number of modules in the final product that are not reused modules. A module is considered reused if it was developed for another project and has less than 20% of it code changed.

In the testing and debugging areas, other physical output metrics that have been measured in commercial practice include:

Number of Fault Reports Generated -- Number of physical forms produced from testing, debugging, review or inspection activities.

Number of Fault Reports Closed -- Number of reported faults solved satisfactorily as indicated by a "signed off" form. On the other hand, "source statements per person year" that have been consumed in large projects provide an overall measure of productivity if an organization is able to take advantage of the statistical Law of Large Numbers, such as reported in [VOSBURGH-84]. When applied consistently over time (at least three years), such rough measures can be valuable for establishing baselines against which evaluations can be made of various factors which are suspected of having an impact on productivity. This type of factor analysis is what is most important when seeking means for productivity improvement.

Some of the factors that have been considered important as having an impact on software development productivity are as follows:

Software Complexity -- (No single measure exists. See below)

Software Quality -- (No single measure exists. See below)

Degree of Use of Modern Programming Practices -- (No single measure exists. Largely subjective)

Software Support Environment -- (No single measure exists. Largely subjective)

Total Effort -- Number of person months of effort used on a project, starting when the requirements and specifications become final through acceptance testing. It includes programming effort plus managerial and clerical overhead. (Definitions vary)

In summary, the state of practice in measuring software productivity is not very far advanced. Only those organizations that have been collecting a large number of well-defined statistics for many years have been able to establish a data base that can be used for comparative analyses (for example, IBM, TRW and ITT).

Category A2

Software Development Productivity (Actual)
(Functional Output per Unit Input)

Another important way to view software life cycle progress with respect to output divided by input is from a user or customer perspective. That is, measurements are taken to help answer the question: How "productive" is the software group in satisfying user requirements? The difficulty with this question is its inherently subjective nature. In essence it is a marketing or sales question best answered by user satisfaction questionnaires or inspection of available sales volume information.

The reason for increasing interest in this measurement approach is due to incredibly large backlogs of user requests for application software systems in the commercial business, Government information, and Defense logistics support communities. Measurements in this area are aimed at demonstrating results of actions taken to increase the output of software support organizations as a whole.

In spite of its mathematical intractability, many have tried to measure this area, but with little success to date. The most effort to date has probably been expended by IBM. Their approach has been used to support internal administration of data processing operations for the past 4-5 years [IBM-82]. Recently, it has been "discovered" by outsiders and is finding its way informally into some IBM customer data processing organizations.

As with physical outputs, a clear definition is required for the work product that can be divided by the labor input. IBM have dubbed their work product measure from this customer or function-delivered perspective as "function points," and recommend this as the primary measure of applications development and maintenance work product.

The function points metric sizes an application program by quantifying the data handling implied in five major forms or elements [Due to the extreme size of the IBM document, only portions are extracted to give the reader a flavor for its content]. User External Inputs -- IBM provides an elaborate full-page definition that describes what to count and what not to count. For example, input forms, scanner forms, terminal screens, keyed input and transactions from another application are counted after first being classified as being "simple," "average," or "complex." Such things as "inquiry transactions are not counted since these are counted "as inquiries in a later question."

User External Outputs -- Again, this requires a full page definition from IBM [which is too long to quote here]. Examples of what to count which must be classified are: Printed reports, terminal printed output, terminal screens, operator messages, transactions to another application. Caveats are included such as "do not count output terminal screens that are needed by the system only because of the specific technical implementation."

User External Inquiries -- Another lengthy IBM definition, only part of which is to count each input/output combination where an on-line input generates and causes an immediate on-line output. Data are entered only for control purposes, that is, to direct the inquiry search. No update is involved in an inquiry...but, do not confuse a major query facility as an inquiry, etc.

User Logical Master Files -- Count each major logical user data group. This count should include each unique machine-readable logical file, or, within a data base, each logical grouping of data from the viewpoint of the user, that is generated, used, or manipulated by the application. For example: Customer record, part master file, customer cross-referenced index. These must be classified as done with other elements above. [Definition continues]

Interfaces to Other Systems -- Count all major machine-readable interfaces to other applications that do not consist of transactions. Files, consisting of records, shared between applications should be counted within each application. They should be counted as files or interfaces, but not both. Interfaces, consisting of transactions, should have been handled as inputs and outputs. Interfaces to other systems should be classified for complexity with the definitions used to classify user logical master files.

After these difficult to define measures are attained, IBM further recommends "adjusting the result" by applying 14 general application characteristics, such as:

Performance -- The applicaton performance in either response time or throughput is a consideration in the design, implementation, and maintenance.

Transaction Rate -- The transaction rate is high and had influence on the design, implementation, and maintenance of the application.

Ease of Installation -- Conversion and installation ease were incorporated in the design and implementation. A conversion and installation plan was provided and it was tested during system test.

Internal Processing Complexity -- Internal processing is complex in this application. An application is complex if there are many interactions and decision points and extensive logical or mathematical equations. It is also complex if it has a preponderance of exception processing resulting in many incomplete transactions that must be resolved later or again.

Each of these "general application characteristics" provided a number from zero to five that represents the degree of influence that each had on the value of the application to the user based upon the following definitions:

0 -- Not present, or no influence

1 -- Incidental influence

2 -- Moderate influence

3 -- Average influence

4 -- Significant influence

5 -- Strong influence throughout

It should be quite obvious that this approach, although conceptually a proper thing to do, is fraught with subjectivity and should be only considered as experimental at best. Much research is required to solve this category of software productivity measurement to make it useful for STARS mission critical systems software, even when in the logistics support phase.

Category B1b

Software Product Attributes (Actual) (In-Process Software Quality)

This area of software metrics has received a great deal of attention in the academic and research world. In-process software quality can be broken up into the various manifestations or translations of software during its life cycle to include requirements specifications, software designs, source code, object code, run-time code and associated documentation. Each of these categories of software has received the attention of researchers attempting to identify metrics to measure its quality.

One of the earliest papers on measuring program quality was by Rubey and Hartwick in 1968 [RUBEY-68]. Recently, Troy investigated measures to aid in the evaluation of software designs [TROY-82]. Major investigations of software quality from a Department of Defense perspective, such as reported in [MANLEY-76] and [RICHARDS-76], discovered more problems than solutions...a situation that still exists today.

In short, there is a very large literature on software quality metrics, but no generally accepted metrics that have been put into widespread practice.

Category B1d

Software Product Attributes (Actual) (Software Testability)

The goal of this area of software metrics is to help researchers and practitioners find ways to make it easier to test software and to determine to what extent software has indeed been tested. This area is still in the research stage of development. See [MONHANTY-76] for one of the earliest definitions of the problem and a proposed solution. Measures involve "accessibility," "testability," and "testedness" which attempt to quantify, for example, the relative ease of testing existing software and the extent of testing that has been accomplished by specific test cases.

Category B2b

Software Product Attributes (Actual) (Software Quality Upon Delivery to Customer)

There are almost as many definitions of software quality as there are researchers. The reason is the ambiguous nature of the term. Several overall perspectives are important to mention:

Microscopic versus macroscopic -- This is an internal versus an external viewpoint. By internal or microscopic is meant simply a rating given the software with respect to how well it meets its detailed requirements specification. By external or macroscopic is meant how well the software satisfies the user in operation, regardless of whether or not it perfectly meets the specification.

Quality Attributes -- This is a definitional perspective with respect to what is included within the "quality" umbrella. Does quality include "reliability," "portability," "robustness," "maintainability," "ease of use," "ease of learning," etc., or only one or a combination of these attributes that may or may not be associated with software products?

Underlying Theory -- Scholars working in the software quality metrics field have based their work on a wide variety of mathematical foundations. These include such things as Shannon's theory, Zipf's law, Halstead's metrics, reliability models of Jelinski-Moranda, Shooman, Littlewood-Verrall and others. A good summary of these diverse approaches can be found in [MOHANTY-79].

From a pragmatic point of view, a major effort was undertaken by the Department of Defense to try to understand software reliability as a major component of software quality from the customer's perspective. Two volumes of [MANLEY-76] are devoted to a state-of-the-art explanation of software reliability to include generally accepted mathematical definitions that are based upon the probability of software failing (or not failing) while in use. This work provided the foundation for major software inititatives that manifested themselves in DoD Directive 5000.29 and subsequent implementing regulations and standards.

Concurrent to the mid-1970's DoD initiative, the commercial industry business software sector reasoned that the most generally accepted and useful metric was simply the number of "problems" or "faults" or "defects" that can be counted at predetermined points during the software life cycle. This rather simplistic approach on the surface has proven quite useful in very large organzations such as IBM [JONES-78], and ITT [VOSBURGH-84] where large empirical statistical data bases can be developed relatively easily.

In essence, this approach assumes that software quality is synonymous with a presence or absence of software problems, much as we subjectively view our family automobile or service in a restaurant.

In short, the universally-acceptable software quality metric does not exist. Nor will it in the foreseeable future. This does not mean that this most important feature of software should not be measured. On the contrary, this implies that further research must be devoted to narrowing the choices of offerings so that a metric, or set of metrics encompassing both the user and developer viewpoints , can be used to evaluate and compare software products developed for mission critical systems.

Category B1c/B2c

Software Product Attributes (Software Complexity Measures)

A measure of potential computer program complexity is important for estimating the total cost and schedule of a software development effort. Such measures are used as inputs to cost estimating models such as SLIM, Wolverton, Schneider or PRICE-S [MOHANTY-81]. However, it is one of the more difficult areas of software metrics to devise solutions. One major reason is the lack of an adequate empirical data base.

To develop a software complexity data base, measures must be taken of completed programs and then judgments made as to which measures of complexity seem to best fit the perceived complexity of the existing program. Once this difficult task is completed, then and only then, can an organization hope to predict the complexity of yet to be developed software systems. As a result, this area of software metrics is still in a research stage [and probably will continue to be for at least 5-10 more years].

A paper presented at the 7th ICSE [ELSHOFF-8[illegible]] summarized the state of the art in complexity metrics fairly well. Twenty complexity measures were selected and studied for how well they identified the more complex procedures in a software system:

The results of Elshoff's experiments indicated that the following subset of six numbered and rank ordered measures accounted for over 90 percent of identified difficult programs in the study. The first four alone can be useful in identifying programs that are abnormally complex and also help guide steps to reduce their complexity [NOTE: Only after the programming is complete].

(1) Length -- The length of the program is the total number of tokens, the sum of the total operators and total operands.

(2) Unique Operators -- Unique operators is a count of the number of unique tokens that are language keywords or symbols, the number of function references, and the number of unique labels that are the targets of GO TO statements.

(3) Data Difficulty -- The data difficulty is the average number of appearances of each operand. It is computed as the total operands divided by the unique operands.

(4) Unique Operands -- Unique operands is a count of the unique variables and constants.

(5) Statements at Level 10 -- A count of the statements that are nested 10 or more levels deep. (Arbitrary since 10 percent of the statements Elshoff was counting were 10 levels deep).

(6) Identifiers -- The identifiers that are explicitly declared are counted including those variables that are declared but never used.

Source Lines -- The number of lines is counted including all comment lines and blank lines.

Input Lines -- The number of lines after expansion by the macro preprocessor is counted. This is the number of input records read to compile the program.

Statements -- The HOL statements are counted including non-executable statements such as DECLARE in PL/1

Predicates (first known as cyclomatic complexity [MCCABE-76]) -- The predicate count, is the number of execution paths from the entry point to the exit point of the program.

Conditions -- Conditions are similar to predicates with the condition of counting logical operators in the predicates. For example, the single predicate, (A<B)&(C>D), is counted as two conditions since the falsity of either the condition A<B or the condition C>D leads to the falsity of the predicate.

Blocks -- This is a count of the number of PROCEDURE, BEGIN, DO, IF, ON and SELECT statements. These statements map flow of control and nesting levels in PL/1.

Call Statements -- The CALL statements are counted.

Total Operators -- Total number of occurrences of the unique operators.

Total Operands -- Total number of occurrences of the unique operands.

Vocabulary -- The vocabulary is the sum of the number of unique operators and unique operands, i.e., the unique tokens (words and symbols).

Volume -- The volume is a measure of the minimum number of bits needed to represent a program. Using the vocabulary and a frequency count of each of the words in the vocabulary, a Huffman encoding can generate the minimum bit representation. Analysis of Huffman encoding leads to the formula for volume which is:

$$\text{Volume} = \text{Length} \times \log_2 (\text{Vocabulary})$$

Difficulty -- The difficulty is defined as one-half the product of the unique operators and the data difficulty.

Understanding Effort -- The understanding effort equals the volume times difficulty.

Construction Effort -- Halstead proposed construction effort as a measure of the amount of work that was required to write a program. The formula is complicated [HALSTEAD-77] as well as controversial [LASSEZ-82].

Category C

Predictive Software Metrics

Software estimating models are very important to everyone involved in automated systems development and support. For example, software management is vitally interested in accurately estimating software project costs and overall schedule. Technical management is interested in estimating such things as hardware requirements to support a planned software system with respect to memory size, operating speed, and so forth. Some of the earliest work in software metrics was in this area. For example, see [LABOLLE-66 and NELSON-67].

As a prerequisite for success, a high quality statistical data base of validated metrics is required as inputs into any predictive model. Therefore, many others have experimented with correlations of factors with various models to determine optimum combinations of metrics as inputs to a specific algorithm to obtain a useful output, i.e,. analytical models such as [WOLVERTON-72, SHOOMAN-76 and PUTNAM-78].

REFERENCES

FARR-64: Farr. L., and Zagorski, H.J., "Factors that Affect the Cost of Computer Programming. Volume II: A Quantitative Analysis." System Development Corporation, Technical Documentary Report No. ESD-TDR-64-448, September 1964

LABOLLE-66: LaBolle, V., "Development of Equations for Estimating the Cost of Computer Program Production," System Development Corporation, Santa Monica, CA, 1966

NELSON-67: Nelson, E. A., "Management Handbook for the Estimation of Computer Programming Costs," TM-3225/000/01, System Development Corporation, Santa Monica, CA 1967.

RUBEY-68: Rubey, R. J. and R. D. Hartwick, "Quantitative Measurement of Program Quality," Proceedings ACM National Conference, 1968, pp. 671-677.

HALSTEAD-72: Halstead, M. H., "Natural Laws Controlling Algorithmic Structure?," SIGPLAN Notices, Vol. 7, No. 2, Feb. 1972

WOLVERTON-72: Wolverton, R. W. and G. J. Schick, "Assessment of Software Reliability," TRW Software Series TRW-SS-73-04, Sep 1972.

BOEHM-73: Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

MANLEY-75: Manley, J. H. and M. Lipow, "Findings and Recommendations of the Joint Logistics Commanders' Software Reliability Work Group," Vols. I and II, National Technical Information Service, Department of Commerce, Springfield, VA, Nov 1975.

BOEHM-76: Boehm, B. W. et al, "Quantitative Evaluation of Software Quality," Proceedings, 2nd International Conference on Software Engineering, Oct 1976, pp. 592-605.

FAGAN-76: Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 182-211.

MCCABE-76: McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

MONHANTY-76: Monhanty, S. N. and M. Adamowicz, "Proposed Measures for the Evaluation of Software," Proceedings of the Symposium on Computer Software Engineering," New York, 1976, Polytechnic Press, Polytechnic Institute of New York, Brooklyn, N.Y. 1976

RICHARDS-76: Richards, P.K., et. al., "Factors in Software Quality," General Electric Presentation under RADC Contract F030602-76-C-0417, 1976

SHOOMAN-76: Shooman, M. L., "Structural Models for Software Reliability Prediction," Proceedings 2nd International Conference on Software Engineering, 1976, pp. 268-280.

BASILI-77: Basili, V. R., et al., "The Software Engineering Laboratory," Technical Report TR-535, Department of Computer Science, University of Maryland, May 1977.

HALSTEAD-77: Halstead, M. H., Elements of Software Science, Elsevier North-Holland, New York, 1977, 127 pp.

WALSTON-77: Walston, C. E. and C. P. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, Vol. 16, No. 1, pp. 54-73, 1977.

PUTNAM-78: Putnam, L. H., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering, SE-4, 1978, pp. 345-361.

JONES-78: Jones, T. C., "Measuring Programming Quality and Productivity," IBM Systems Journal, Vol. 17, No. 1, 1978.

MOHANTY-79: Mohanty, S. N., "Models and Measurements for Quality Assessment of Software," Computing Surveys, Vol. 11, No. 3, pp. 251-275, Sep. 1979.

BASILI-81: Basili, V. R. and K. E. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," The Journal of Systems and Software, Vol. 2, No. 1, pp. 47-57, 1981.

MOHANTY-81: Mohanty, S. N., "Software Cost Estimation: Present and Future," Software-Practice and Experience, Vol. 11, 1981, pp. 103-121

LASSEZ-82: Lassez, J.-L., D. van der Knijff and J. Shepherd, "A Critical Examination of Software Science," Journal of Systems and Software, Vol. 2, No. 2, June 1981, pp. 105-112.

TROY-82: Troy, D. A. and S. H. Zweben, "Measuring the Quality of Structured Designs," Journal of Systems and Software, Vol. 2, No. 2, June 1981, pp. 113-120.

IBM-82: "Application Development and Maintenance Measurement and Analysis," IBM Corporate Information Systems and Administration Guideline, No. 303, Feb. 26, 1982

PANZL-82: Panzl, D. J. "A Method for Evaluating Software Development Techniques," Journal of Systems and Software, Vol. 2, No. 2, June 1981, pp. 133-137.

ELSHOFF-84: Elshoff, J. L., "Characteristic Program Complexity Measures," Proceedings of the 7th International Conference on Software Engineering, IEEE Computer Society, Silver Spring, Maryland, 1984.

VOSBURGH-84: Vosburgh, J., et al., "Productivity Factors and Programming Environments," Proceedings of the 7th International Conference on Software Engineering, IEEE Computer Society, Silver Spring, Maryland, 1984.

AFR 800-14 HISTORY

Dr. John H. Manley

1969-72	Problem with lack of policy and specific guidance for software contained in Air Force "weapon systems" recognized and documented by AFSC. Initial milestone was 9 volume Mission Analysis, CCIP-85, sponsored by Hq AFSC and chaired by Barry Boehm, then at Rand Corporation.
Aug 73	AFCS established Assistant for Processor and Software Planning [forerunner of XRF and now ____]
Sep 73	Initial three-page "strawman" of AFR 800-XX developed by Hq AFSC -- desired principles
Oct 73	Draft strawman 800-XX sent to Air Staff for comment
Nov 73	"Embedded Computer Systems" defined and categorized as class of systems that would be the subject matter of 800-XX [reported publicly for first time at NCC 1974 in Chicago]
Dec 73	Formal work begun on "AFM 800-XX" due to favorable response by Air Staff -- How-to-do-it instructions for AFR 800-XX policy guidance
May 74	AFR 800-14 policy guidance published by USAF
Nov 74	Final draft of AFM 800-XX completed by AFSC
Jan 75	AFM 800-XX redisignated AFR 800-14, Vol. 2 and the original AFR 800-14 redisignated AFR 800-14, Vol. 1
Sep 75	AFR 8001-14, Vols. 1 and 2 officially published.

FORMAL SOFTWARE VERIFICATION AS AN EXAMPLE OF SOFTWARE TECHNOLOGY TRANSFER

Ann B. Marmor-Squires
TRW Defense Systems Group
2751 Prosperity Avenue
Fairfax, VA 22031

May 1984

ABSTRACT

Formal software verification has had an active history of over twenty years of research, development and application. It is currently in use in varying degrees in the development of trusted computer systems and critical sections of systems demanding very high reliability. Its history is traced in this paper as an example of software technology transfer. A formal verification chronology is given and several observations on the technology transfer process, based on the formal verification experience, are made.

1.0 INTRODUCTION

Formal software verification refers to a rigorous mathematical proof of correctness with respect to a set of requirements. The concept of developing software programs that can be mathematically proved correct may be traced back to the early days of computing, particularly in some of Turing's writings. The active theoretical work in proving programs correct began in the early to mid-1960s. By the mid-1970s, prototype automated program verifiers and interactive verification systems were developed. Early experimental use of the systems by groups other than the developers was carried out in the mid-to-late 1970s. From the late 1970s on, formal software verification has become an important component in the development of certain critical software, particularly in systems having very high reliability and security requirements. In addition, there have been several offshoots of formal software verification that are in use by software practitioners--symbolic execution, rigorous program reasoning and "desk checking". While the maturation and transfer of formal software verification is not yet complete (it is not yet in widespread commercial use), it does provide a good example of the software technology transfer process. The chronology of formal verification, given in Section 3, provides a scenario for transferring modern software technology into practice that is fairly typical. The experiences in the research, development and transfer of formal verification also provide some insight into what can facilitate or inhibit technology transfer. These are the subjects of Section 4.

This case history of formal software verification as an example of software technology transfer was facilitated by the availability of several survey articles and textbooks on the subject. However, none of these provided a critical assessment of the technology nor of its transfer into practice. The author's experience with formal verification technology and private communication with the verification system developers as well as users provided an evaluation of the technology transfer process and a measure of its success.

2.0 A SHORT SYNOPSIS OF FORMAL SOFTWARE VERIFICATION

Formal software verification may be defined as a rigorous proof of correctness between levels of system abstraction. In the realm of formal verification, "correctness" refers to consistency between the levels shown in Figure 1 (abstracted from [1]). The requirements are the principal system concepts with which the high order language implementation (the program) needs to be consistent. The formal model and formal specifications represent different levels of abstraction of the properties listed in the requirements. The model and specifications are written in a formal mathematical notation and describe system characteristics. Correctness, then, is the consistency that is maintained between the formal model and the formal specification, as well as between the formal specifications and the implementation.

The goal of formal verification is to demonstrate mathematically that the implementation is correct with respect to its requirements. This is done in a progression of steps that show consistency throughout the intervening levels (Figure 1). Proofs of consistency between all adjacent levels of abstraction is equivalent to directly establishing the correctness of the implementation with respect to the requirements.

The formal software verification process is supported by automated (interactive) specification and verification tools. Some current verification systems are intended solely for proofs of consistency between specifications, while others include implementation languages and tools

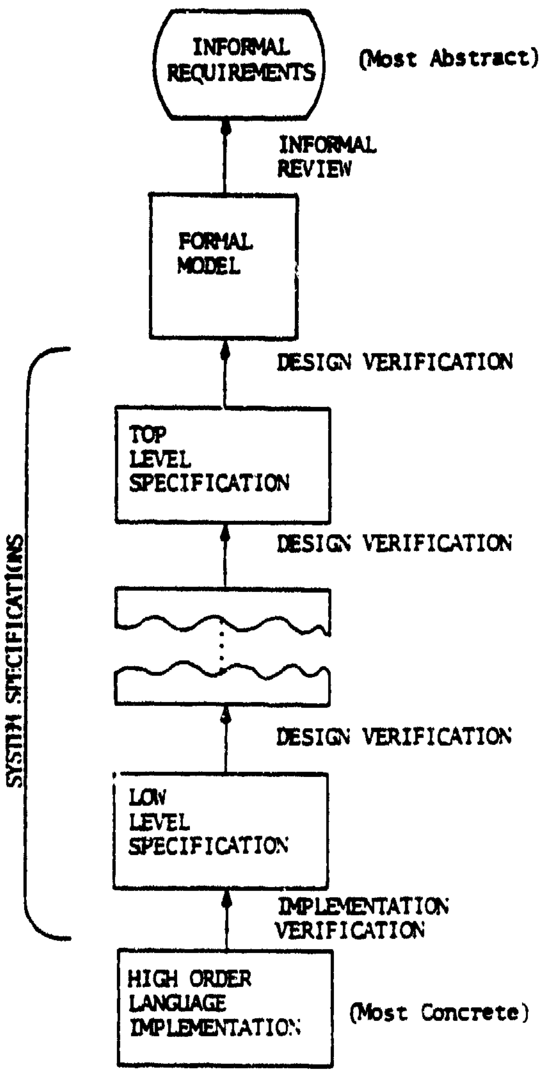


Figure 1. Formal Software Verification Process

to prove properties about an implementation. The paper by Cheheyl, et. al. [2] is a survey of several of the current verification environments.

Most current verification environments that support implementation verification focus on three main tools: a parser, a verification condition generator and a theorem prover. The parser checks specifications and implementation for syntactic and semantic correctness. The verification condition generator produces theorems that must be proved in order to show the consistency of the implementation with the lowest level specification. The theorem prover is used to prove the conditions that are generated. Theorem prover capabilities vary from fully automatic to merely doing "proof checking"--checking that the user has proceeded along a legal chain of logical reasoning.

3.0 FORMAL SOFTWARE VERIFICATION CHRONOLOGY

The activities and milestones pertinent to tracing the transfer of formal software verification into use on "real" systems are:

- 1958-1966: Some early work is done on the theoretical foundations of the equivalence of logical program schemes, algorithms and program statements. Ianov's work in Russia in the late 1950s on the theory of program schemes [3], McCarthy's presentation at IFIP Congress in 1962 on a mathematical science of computation [4], Igarashi's Ph.D. thesis in Japan in 1964 on algorithm and program statement equivalence [5], and Naur's work in 1966 on program proof by general snapshots [6] are representative of the early theoretical work.
- 1967-1969: Floyd's 1967 paper on assigning meanings to programs [7] is the one most authors cite as the basis for work in program verification. Floyd introduces the inductive assertion technique. Dijkstra's 1968 paper presents a constructive approach to program correctness [8], Manna researches the relationship between the correctness of programs and first order predicate calculus [9], and Hoare presents an axiomatic basis for proofs of program properties [10]. Work in automated theorem proving begins.

- 1969-1971: Further theoretical work on theorem proving, the correctness of non-deterministic programs, parallel program schemes, program termination, program synthesis and program scheme equivalences is carried out. Proofs of programs (FIND, TREESORT, QUICKSORT) appear in the literature.

- 1969-1972: The first few Ph.D., dissertations on program verifiers are completed: King's thesis at Carnegie-Mellon University [11], Good's thesis at the University of Wisconsin [12], and Gerhart's thesis on verifying APL programs at Carnegie-Mellon [13].

- 1972: Hoare publishes a paper on the proof of correctness of data representations [14].

- 1972: Several survey articles are published on program verification: Elspas' Computing Surveys paper [15], Linden's presentation at the Fall Joint Computer Conference [16] and London's presentation at the ACM National Conference [17] are the major ones.

- 1972: The computer security community identifies the need for demonstrating the correct operation of a security kernel as an important part of the solution to building secure computer systems.

- 1973-1974: Ragland's thesis on the construction and verification of a verification condition generator based on the inductive assertion technique appears [18]. Good, London and Bledsoe report on the development of the XIVUS interactive program verification system [19] and Elspas, Levitt and Waldinger describe another program verification system [20]. The former is the forerunner of the University of Texas Gypsy verification system and the latter developed into the SRI verifier.

- 1973-1975: The axiomatic definition of the programming language Pascal is published by Hoare and Wirth [21]. Boyer and Moore report on proving theorems about LISP programs [22]. The design of the verifiable language, Gypsy, begins at the University of Texas [23] and a committee is formed to develop Euclid, a verifiable programming language. SPECIAL, a formal specification language is developed at SRI [24] and the Ina Jo specification language is developed at System Development Corporation [25]. Reports on the languages were published several years after the work was begun.

- 1973-1977: SPECIAL is further developed as part of SRI's work on the Hierarchical Development Methodology (HDM), a specification and verification system for Gypsy programs is developed and the Ina Jo methodology, later to be called the Formal Development Methodology (FDM), is under development. All of these formal verification activities are a result of the need for formal verification as a major component in assuring that computer systems are secure.

- 1974: The first textbook is published on the subject of formal software verification [26].

- 1975-1979: The early application of formal software verification and the experimental verification systems is actively pursued. SRI's HDM is applied to the development of PSOS, the Provably Secure Operating System [27], Gypsy and its verification environment are applied to communications processing examples [28] and FDM applications are begun. All the verification languages (specification and/or implementation languages) and their verification systems are early prototypes and are considered experimental.

- 1977-1980: Work on the Kernelized Secure Operating System (KSOS), a "secure version of the UNIX^{tm} operating system" is undertaken. This is one of the first major applications of verification technology. The project runs into difficulty because the technology is not mature enough for such an application. SRI's HDM (and SPECIAL language) is used by Ford Aerospace (not the original developers) on the KSOS Project. Another major application of formal verification to security applications is the development of KVM-370, a kernelized version of the IBM VM-370 operating system. These applications, as well as use of the Gypsy system, provide feedback to the major verification system developers.

- 1979: Boyer and Moore publish A Computational Logic [29].

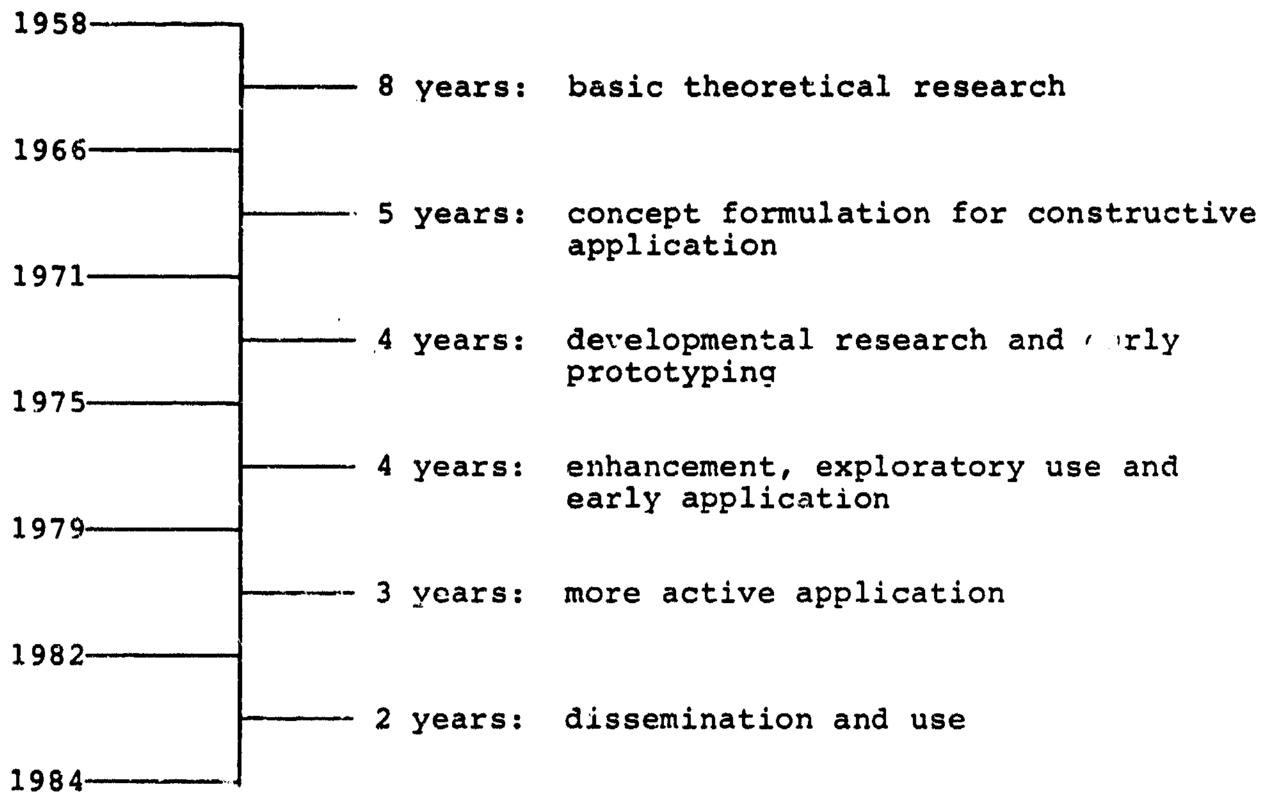
- 1979: Gerhart reports on the development of the AFFIRM System, an outgrowth of the XIVUS system combined with work on abstract data type specification and verification [30]. Luckham describes work undertaken on the Pascal verifier [31].

- 1980-1982: Application of verification technology to the development of secure operating systems, trusted downgrade processors and communications processors as well as to a fault tolerant computer for aircraft control (NASA's SIFT system) is pursued. The major experimental verification systems (the HDM tools, the Gypsy environment and the FDM tools) are used actively (although still experimentally) by groups other than the developers--by major computer manufacturers, large defense contractors and by various government laboratories. In 1980, the DoD Computer Security Initiative that is under way has the Trusted Computer System Verification Technology R&D Program as a major thrust aimed at the practical widespread application of verification technology for trusted system development. Initial week-long courses on verification technology are given.

- 1983-present: The DoD Computer Security Center, established in 1981, issues the "Trusted Computer System Evaluation Criteria" [32]. It includes verification technology as a

^{tm}UNIX is a trademark of AT&T Bell Laboratories major method for satisfying the trusted computer system assurance requirements for achieving high ratings according to the criteria. The stabilization of the major verification systems is under way and it is expected that they will be production-quality in approximately 1986. Although Gypsy and HDM currently run on the DEC-10 and DEC-20 and FDM runs on IBM 360, all will be available on the Multics at the DoD Computer Security Center for use by industry. They will also be ported to run on the Symbolics 3600 which will make them more widely available. A verification system is under way for Euclid at I.P. Sharpe and research on a verifiable subset of Ada® has begun.

This chronology indicates that formal software verification has had the following periods during its history:



1958
1966
1971
1975
1979
1982
1984
8 years: basic theoretical research
5 years: concept formulation for constructive application
4 years: developmental research and early prototyping
4 years: enhancement, exploratory use and early application
3 years: more active application
2 years: dissemination and use

The phase from about 1958 to 1966 represents the early work of basic theoretical research in proof of correctness. During the next five years, 1966 to 1971, theoretical work continues but the practical application of the theory is explored. In the late 1960s and early

®Ada is a trademark of the Department of Defense.

1970s, several verifiers are implemented. From 1971 to 1975, developmental research on specification and verification languages as well as early prototyping of verification systems takes place. In 1972, three survey articles appear and by 1974, a textbook on program verification is published. The phase from 1975 to 1979 is major enhancement to the languages and verification systems, as well as exploratory use on small to medium scale applications primarily by the developers. From 1979 to 1982 more active application of formal software verification is pursued. Since approximately 1982, the technology has been disseminated and used, primarily for critical security applications.

The chronology of formal software verification seems to be fairly typical of the process of technology transfer. The underlying theoretical foundation and formulation of concepts, though, probably took a longer time than is typical.

4.0 SOME OBSERVATIONS

The history of formal software verification provides several examples of what can reasonably be expected to happen in the process of transferring software technology into actual use. These are discussed in this section.

As the chronology illustrates, there was significant and lengthy (over ten years) activity on the theoretical foundations of formal software verification before the first program verifier was developed as a university research project and subsequent Ph.D. thesis. In fact, theoretical work in the field is still on-going to expand what program and system properties one is able to formally specify and verify. It has been a major undertaking to transfer formal software verification to practical (but not yet widespread) use. There are several reasons for this:

- Formal software verification is hard. It is a labor-intensive activity requiring mathematically sophisticated users. Many universities that have a software engineering curriculum do not include verification as a major component and thus not many students are exposed to verification technology, particularly at the undergraduate level.
- After the theoretical foundation for verification was sound, several methodologies developed that incorporated formal verification, namely, HDM, FDM, Gypsy and AFFIRM. As the methodologies stabilized, languages were developed for specification and implementation to be incorporated into the methodologies. Automated support tools to effectively use the methodologies and languages, however, have lagged behind. Without automated assistance, the application of verification technology is a tedious and error-prone activity. The current generation of automated verification systems are experimental ("prototype"); however, they have been used in several applications, particularly for trusted computer system development.
- Formal software verification is a computation-intensive activity. Most of the automated verification systems run as large LISP programs on large mainframes: currently the DEC-20 (FDM tools run on large IBM hardware). In order for an enterprise, e.g., government laboratory, computer manufacturer or defense contractor, to effectively use the technology, a major capital investment is needed.
- Formal software verification is costly. Since it has been labor-intensive and computation-intensive, it has been a

costly process to apply. Cost estimates have ranged from 2-3 times "commercial practice" software development costs, although these estimates often do not take into account the full life-cycle development costs for "commercial practice". The benefits of verification technology are in ease of maintenance and in correct operation--these are often difficult costs to measure.

- Early practical application of formal software verification did not produce convincing demonstrations of computer systems that could be effectively used. The KSOS and PSOS applications tried to use verification technology before it was ready for such ambitious projects. The experience on these early applications, however, provided invaluable feedback from the users to the developers on improvements needed in all areas: methodology, languages and automated support tools.

The reasons cited above illustrate some of the hindrances to effective technology transfer. However, other activities in the chronology of formal software verification significantly helped move the technology from university research toward practical application. In the early 1970s, the computer security community focused on a new approach to building trusted computer systems that incorporated verification technology. The DoD recognized a need for convincingly demonstrating that computer systems can be built to satisfy rigorous security requirements. Verification technology is a major technique for carrying out this convincing demonstration. The emphasis that DoD has put on solutions to the computer security problem, in general, and on making verification technology practical, has had a significant impact on the transfer of verification technology. DoD has provided the major funding for the verification methodologies, languages and tools. A notable exception is Ina Jo/FDM, which was developed by System Development Corporation using internal funds. It is fair to say that without DoD taking the lead in pushing verification technology, it would have remained primarily a university activity and produced mainly theoretical results, rather than prototype system implementations.

In addition to funding the verification technology developments, DoD also funded the initial major applications. These were high-risk projects that a single enterprise would probably not have undertaken. Although the projects attempted to use verification technology before it was ready for such ambitious projects (e.g., the language compilers were not completed, the verification tools were early implementations), the feedback to the developers was invaluable. The experience also helped focus the DoD activities on those aspects of verification technology that were needed to make it practical and effective.

DoD continues to take the lead in application (and funding) of verification technology. The activity currently focuses on stabilizing the major verification systems and making them widely available, porting the systems to machines that are less costly and, hopefully, more widely available as well as less computation-intensive (e.g., Symbolics 3600), and providing high-quality training, friendly user interfaces to the systems and good user documentation.

High-quality training and good documentation on specific systems is critical. Formal verification is a radically new technology, particularly for software developers who are not recent university graduates. This training must be effective in spite of the initial hostility that new users will have towards the new technology.

There have also been additional side benefits resulting from the software verification research and development. While formal verification fully supported by automated interactive tools has been slow in getting into the software practitioner's daily work world, valuable side effects have had broader impact in a shorter time period. The foundations of techniques such as error-free program development, symbolic program execution as well as rigorous reasoning and "desk checking" of programs during development can be traced back to formal verification technology. These techniques have become a more widespread part of the practitioner's set of useful techniques for software developments.

5.0 REFERENCES

1. B. Hartman, M. Taylor and A. Marmor-Squires. "Formal Verification in the Trusted System Evaluation Process", IEEE EASCON '82 Proceedings, 1982.

2. M. Cheheyl, M. Gasser, G. Huff and J. Millen. "Verifying Security", ACM Computing Surveys, Volume 13, Number 3, September 1981.

3. I. Ianov. "On the Equivalence and Transformation of Program Schemes", Communications of the ACM, Volume 1, Number 10, October 1958.

4. J. McCarthy. "Toward a Mathematical Science of Computation", IFIP 62 Proceedings, 1962.

5. S. Igarashi. "An Axiomatic Approach to the Equivalence Problems of Algorithms with Applications", U. of Tokyo, Ph.D. Thesis, 1964.

6. P. Naur. "Proof of Algorithms by General Snapshots", BIT Volume 6, Number 4, 1966.

7. R. Floyd. "Assigning Meaning to Programs", Mathematical Aspects of Computer Science, Volume 19, American Mathematics Society, Providence, RI, 1967.

8. E. Dijkstra. "A Constructive Approach to the Problem of Program Correctness", BIT Volume 8, 1968.

9. Z. Manna. "Properties of Programs and the First Order Predicate Calculus", Journal of the ACM, Volume 16, Number 2, April 1969.

10. C. Hoare. "The Axiomatic Basis for Computer Programming", Communications of the ACM, Volume 12, Number 10, October 1969.

11. J. King. "A Program Verifier", Carnegie-Mellon University, Ph.D. Thesis, September 1969.

12. D. Good. "Toward a Man-Machine System of Proving Program Correctness", University of Wisconsin, Ph.D. Thesis, 1970.

13. S. Gerhart. "Verification of APL Programs", Carnegie-Mellon University, Ph.D. Thesis, November 1972.

14. C. Hoare. "Proof of Correctness of Data Representations", Acta Informatica 1, 1972.

15. B. Elspas, K. Levitt, R. Waldinger and A. Waksman. "An Assessment of Techniques for Proving Program Correctness", Computing Surveys, June 1972.

16. T. Linden. "A Summary of Progress Toward Proving Program Correctness", Fall Joint Computer Conference 72, 1972.

17. R. London, "The Current State of Proving Programs Correct", ACM National Conference, 1972.

18. L. Ragland. "A Verified Program Verifier", University of Texas, Ph.D. Thesis, May 1973.

19. D. Good, R. London and W. Bledsoe. "An Interactive Program Verification System", ISI, Marina del Rey, CA, Technical Report Number 22, September 1974.

20. B. Elspas, K. Levitt and R. Waldinger. "An Interactive System for the Verification of Computer Programs", SRI, Menlo Park, CA, Final Report, 1973.

21. C. Hoare and N. Wirth. "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica 2, 1973.

22. R. Boyer and J. Moore. "Proving Theorems about LISP Functions", Journal of the ACM, Volume 22, Number 1, January 1975.

23. D. Good, R. Cohen, C. Hoch, L. Hunter and D. Hare. "Report on the Language Gypsy: Version 2.0", University of Texas, ICSCA-CMP-10, September 1978. (Earlier version published in 1975.)

24. O. Roubine and L. Robinson. "SPECIAL Reference Manual", SRI Memorandum, August 1976.

25. J. Scheid. "Ina Jo: A Verification Methodology", SDC Report, June 1979.

26. Z. Manna. "A Mathematical Theory of Computation", 1984.

27. P. Neumann, R. Boyer, R. Feiertag, K. Levitt and L. Robinson. "A Provably Secure Operating System", SRI, Menlo Partk, February 1977.

28. D. Good (editor). "Constructing Verifiably Reliable and Secure Communications Processing Systems", University of Texas, ICSCA-CMP-6, January 1977.

29. R. Boyer and J. Moore. "A Computational Logic", NY, Academic Press, 1979.

30. S. Gerhart (editor). "AFFIRM User's Guide", ISI, Marina del Rey, CA, April 1980.

31. D. Luckham. "Stanford Pascal Verifier User Manual", Stanford University, STAN-CS-79-731, 1979.

32. DoD Computer Security Center. "Trusted Computer System Evaluation Criteria", August 1983.

DOD-STD-SDS: The Development of a Standard

R. J. Martin

Georgia Institute of Technology
School of Information and Computer Science
Atlanta, GA 30332

1. Introduction

The following describes the events which have occurred thus far in the development of DOD-STD-SDS on "Defense System Software Development". It is important to note that only the standard is discussed, not the technologies embedded in the standard. Each technology has progressed along its own technology maturation path which will include the technology's appearance in DOD-STD-SDS.

2. The Need for DOD-STD-SDS

DOD-STD-SDS is being developed in response to a variety of problems which have been experienced with respect to the development and acquisition of embedded computer software. In particular, the increasing cost of software, the delivery of software which is not effective in or suitable for operational use, and the extreme difficulty of maintaining software have all contributed to the perceived need for a new standard. It is expected that the use of a joint service standard for software development will reduce the cost and increase the quality of software by allowing industry to implement one set of procedures rather than a different set to respond to each service's set of standards. Furthermore, the employment of consistent terminology and definitions should reduce confusion and facilitate cross-fertilization of ideas and techniques.

3. DOD-STD-SDS Chronology

The following lists the major activities and milestones which have occurred thus far in the development of DOD-STD-SDS.

Sometime prior to April 1979:

In response to problems being experienced in the area of embedded computer system acquisition and maintenance (i.e., escalating costs and inadequate software products), the Joint Logistics Commanders establish the Joint Policy

Coordinating Group on Computer Resource Management. This group, in turn, charters a Computer Software Management Subgroup to examine policies, procedures, regulations, and standards related to defense system software acquisition and make recommendations for the improvement and standardization of that process.

April 1979:

Monterey I, the first Joint Logistics Commanders Software Workshop, is held in Monterey, CA, sponsored by the Computer Software Management Subgroup. The purpose of this workshop is to review the existing policy and guidance to determine areas of conflict and redundancy and to identify shortcomings. Furthermore, the potential for the development of joint service standards is to be investigated. Findings of the workshop are concerned with the inconsistent use of terminology and definitions across the services, as well as the wide assortment of standards. It is further discovered that, in some cases, existing standards include data requirements which should reside in Data Item Descriptions. Recommendations resulting from the workshop include:

"Develop a single unified set of acquisition and development standards for joint service application."

"Define and develop a comprehensive set of Data Item Descriptions (DIDs) for use in software acquisition."

August 1980:

A contract is let for the development of the Data Item Descriptions per the Monterey I recommendation.

At approximately the same time:

The Rome Air Development Center (RADC) lets a contract for the development of a Military Standard on Software Engineering. This is independent of the Joint Logistics Commanders and the Monterey I recommendation.

April 1981:

The RADC contract is changed. Contractor is now tasked to modify MIL-STD-1679 (Navy) on "Weapons System Software Development", MIL-STD-1521A (USAF) on "Technical Reviews and Audits for Systems, Equipments, and Computer Software", MIL-STD-483 (USAF) on "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Software", and MIL-STD-490 on "Specification Practices" to incorporate the new Data Item Descriptions. This effort has now become that recommended at Monterey I.

Mid-year 1981:

The decision is made to write a new standard for software development rather than modify MIL-STD-1679. The new standard is called DOD-STD-SDS, "Defense System Software Development".

June 1982:

Original RADC contract expires. It is replaced by a contract which includes, in addition to the previous effort, the completion of the development of the new Data Item Descriptions.

June - August 1982:

The first version of DOD-STD-SDS, the updated MIL-STD's 1521A, 483, and 490, and the new Data Item Descriptions are released for government and industry review.

October 1982 - January 1983:

Over 5000 comments are received on the document set. Comments range from "change word x to word y" to "start over". Specific changes to DOD-STD-SDS recommended in the comments include: streamline DOD-STD-SDS, require DID preparation, incorporate Ada, simplify software architecture, discuss firmware, emphasize commercial and reusable software, clarify the relationship of the software life cycle to the system life cycle, and modify definitions.

January - May 1983:

Contractor analyzes comments and revises document set.

May - July 1983:

Joint Service/Industry Review Meetings are held. Representatives of major industry organizations participate in a special review meetings which include lengthy discussions with contractor and sponsor concerning resolution of issues and response to comments.

August 1983:

Updated versions of the Military Standards and Data Item Descriptions are available. Another meeting is held with one industry organization which is especially concerned about the new documents.

December 1983:

The "final" revised draft of the document is complete. Of 42 issues raised, 24 are totally resolved; 10 are partially resolved; and 8 are outstanding.

January - March 1984:

Formal review and coordination with government and industry is conducted.

July - December 1984:

Comments from formal coordination are incorporated in standard prior to release.

January 1985:

DOD-STD-SDS is available for use. Work begins immediately to resolve outstanding issues and incorporate lessons learned from initial applications.

4. Summary

The following depicts the phases of the history of DOD-STD-SDS.

```
1979 --+
       |
       +  one year concept formulation
       |
1980 --+
       |
       +  two years initial development
       |
1982 --+
       |
       +  three years review and enhancement
       |       prior to initial release
1985 --+
       |
       +  review and enhancement continues
       |
```

The most striking aspect of the above timeline is the lengthy review cycle. This reflects the fact that if a standard is to be accepted and put into effective use, it cannot be imposed arbitrarily. The potential impact on the organizations both within and without the government must be carefully investigated. This includes an investigation of the maturity and appropriateness of the mandated technologies.

5. References

Briefing Slides from the Joint Service/Industry Review Meeting, 24 May 1983. Prepared by Dynamics Research Corporation.

DOD-STD-SDS, Defense System Software Development, 31 August 1983.

Final Report of the Joint Logistics Commanders Software Workshop, Volume I, Ø1 October 1979.

Private Communication with L. Cooper, 26 June 1984.

Private Communication with R. SanAntonio, 17 February 1984.

STRUCTURED PROGRAMMING: A TECHNOLOGY INSERTION CASE STUDY

Samuel T. Redwine, Jr.

May 1984

1.0 HISTORY

If a technology can be said to trace from a single speech or paper, then for structured programming that paper is Edsger Dijkstra's "Programming Considered as a Human Activity" given at the 1965 IFIP Congress in New York [1]. While it contained virtually all the elements of structured programming, it received much less attention than his 1968 letter to the editor in *Communications of the ACM* more dramatically titled "GO TO Statement Considered Harmful" [2]. (In which, by the way, he noted that concern about the GO TO statement had been discussed in meetings at least as far back as 1959.)

Despite his prior speech and publication in the U.S., the transfer of the concept into the U.S. having the largest impact occurred at an October 1969 NATO Science Committee-sponsored conference in Rome, where Harlan Mills and others from IBM Federal Systems -- there to report on their "Super-Programmer Project" [3] -- heard Dijkstra [4] and carried his ideas home [5]. They proceeded to elaborate and use the ideas on real projects, and report on the results, for example the New York Times Information Bank project on which several papers were published in 1972 [6, 7]. Following these projects, IBM became a major proponent and disseminator of structured programming along with other so-called modern programming methodologies. (To give corporate credit where due, one might note that Dijkstra has been an employee of Burroughs since 1973 although still based in the Netherlands.)

The year 1972 also marked the publication of the first book-length treatment of structured programming by Dahl, Dijkstra, and Hoare [8]. *Datamation* had a laudatory special issue on structured programming in December 1973, and *ACM Computing Surveys* had a special issue in December 1974. A slim textbook was published by Wirth in 1973 [9]; the first traditional undergraduate text was published in 1975 [10]. The first language-specific text for the most widely-used programming language was McCracken's "A Simplified Guide to Structured Programming in COBOL" in 1976 [11].

By 1977, when after 12 years the next IFIP Congress was held in North America, a lot had been written about structured programming. But a survey of 33 large corporations in Los Angeles showed that "only three had fully implemented structured programming along with a well-defined set of procedures and documentation standards for its use." Another 11 "were still experimenting with and evaluating this technique" [12]. So 14 of 33, or 42 percent, might be said to be using it at all. (This survey defined structured programming as essentially structured coding, but 42% also reported using top-down design). A 1979-80 survey in Dallas/Fort Worth showed 40 of 51 (with 12 non-responsors to this question) or 63-78 percent using "structured programming in high-level languages" at all [13].

2.0 DISCUSSION

Structured programming is probably the most famous software-related technology other than high-level languages. It has had virtually universal support in the professional literature and the active support of the largest corporation in computing. Yet it took 12-15 years before half of practitioner organizations had even tried it. Anecdotal evidence suggests that even today, 19 years after the original paper, many organizations have not systematically adopted structured programming and a number do not use it at all.

Mills has suggested one of the problems was its origin in the academic community and the slow transfer out of that community [14]. Ed Yourdan has suggested that "timing" and "packaging" contributed to the delay [15]. These and other aspects such as its essentially mental (vs. tool or product) nature probably all contributed. Part of the resistance may have been because some widely-used programming languages, such as FORTRAN, were awkward to do structured programming in.

The length of time between the first paper and the first text book was eight years; this time might have been easier to shorten than the next six years before a majority of organizations had at least tried it. In addition, the yet another six years or so until it was firmly established in the great majority of organizations might also have been possible to accelerate. Certainly, possibilities for more rapid progress would seem to have existed in all stages.

REFERENCES

[1] E. Dijkstra, "Programming Considered as a Human Activity," *Proceedings of 1965 IFIP Congress,* North-Holland, 1965, pp. 213-17.

[2] E. Dijkstra, "GO TO Statement Considered Harmful," *Communications of the ACM,* Vol. 11, No. 3, March 1968, pp. 147-48.

[3] J.D. Aron, "The 'Super-Programmer Project,'" in Buxton, Naur, and Randell (eds.), *Software Engineering, Concepts and Techniques,* Litton 1976.

[4] E. Dijkstra, "Structured Programming," in Buxton, Naur, and Randell.

[5] H.D. Mills, *Software Productivity,* Little, Brown, and Co., 1983, p. 2.

[6] F.T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal,* Vol. 11, No. 1 (1972), pp. 56-73.

[7] F.T. Baker, "System Quality Through Structured Programming," *AFIPS Proceedings of the 1972 Fall Joint Computer Conference,* AFIPS Press, 1972, pp. 339-44.

[8] O.J. Dahl, E. Dijkstra, and C.A.R. Hoare, *Structured Programming,* Academic Press, 1972.

[9] N. Wirth, *Systematic Programming: An Introduction,* Prentice-Hall, 1973.

[10] C.L. McGowen and R.J. Kelly, "Top-down Structured Programming Techniques," Petrocelli/Charter, 1975.

[11] D. McCracken, "A Simplified Guide to Structured Programming in COBOL," John Wiley & Sons, 1976.

[12] J.B. Holton, "Are the New Programming Techniques Being Used?" *Datamation,* Vol. 23, No. 7, July 1977, pp. 97-103.

[13] L.L. Beck and T.E. Perkins, "A Survey of Software Engineering Practice: Tools, Methods, and Results," *IEEE Transactions on Software Engineering,* Vol. 9, No. 5, September 1983, pp. 541-561.

SDAM/12 January 1984

software design & analysis, inc.

1670 Bear Mountain Drive
Boulder, Colorado 80303

303 499 4782

The Magic Number Eighteen Plus or Minus Three: A Study of Software Technology Maturation

William E. Riddle

ABSTRACT:

It is commonly thought that 10 years is needed for technology to pass from its initial conception into wide-spread use. In the process of gathering data to argue the need for a technology improvement program, we investigated the technology maturation process for three environments supporting software development. Our hypothesis was that more than 10 years was needed; our guess was that the period would be more on the order of 15 years; and we found that it takes on the order of 18 years for systems such as these to mature. Technology maturation time lines for the three systems are presented and some comments are provided on what facilitates and inhibits software technology maturation.

This report was prepared as part of work performed for the Department of Defense STARS Joint Program Office by the Computer and Software Engineering Division of the Institute for Defense Analyses.

1. Introduction

It is commonly thought that it takes 10 years for technology to mature from its initial conception to its wide-spread use. While it is quite hard to pin down exactly when an idea emerges and when it is in wide spread use, this has seemed to be the period for the maturation of technology such as structured programming and Pascal. As a (glib) explanation, it has been noted that this is a typical period of time for a advanced-degree university student to graduate, take a job and get into a position in which he or she can affect the technology used in their organization.

In the process of gathering data to argue the need for a technology improvement program, we investigated the technology maturation process for three environments supporting software development. These three systems were: the Software Requirements Engineering Methodology (SREM) system, the Unix* operating system, and the Smalltalk-80** system.

Our hypothesis was that more than 10 years was needed and our guess was that the period would be more on the order of 15 years. We found that it has taken on the order of 18 years for systems such as these to mature from the initial emergence of underlying concepts to commercialization of a system embodying them. Technology maturation time lines for the three systems are presented and discussed in Section 2.

Partially, the increased length of time is needed because we have considered large systems that consolidate several technologies rather than the individual technologies themselves. And partially, it comes from considering the period to extend up to commercialization. But, there are other reasons that this period is more lengthy than our intuition might have us believe — these are discussed in Section 3.

2. The Software Requirements Engineering Methodology System

SREM was developed by TRW, Huntsville, under the sponsorship of the Army's Ballistic Missile Defense Advanced Technology Center (BMDATC). Active work was begun in mid-1973 and the first version of the system was released in 1977. Since its initial release, SREM has matured past the prototype stage and has been transferred to several government, academic and industrial organizations. It has not as yet been fully commercialized.

2.1. A Short Synopsis of SREM

SREM is a method for developing a formal specification of the data processing requirements for complex, possibly concurrent systems. It provides a set of rules, guidelines and procedures for capturing the requirements in a formal notation called the Requirements Statement Language (RSL). RSL is based on a formal, graph model of data manipulation and provides a medium for describing the required processing in terms of the system's (output) response to various (input) stimuli. RSL allows the specification of performance as well as functionality characteristics.

* Unix is a trademark of AT&T Bell Laboratories.

** Smalltalk-80 is a trademark of Xerox Corporation.

The use of the SREM method is supported by automated tools provided by the Requirements Engineering and Validation System (REVS). The REVS tools allow the syntactic analysis of RSL descriptions, the construction of a requirements database, the analysis of some consistency and completeness characteristics, the development of a simulation model of the system describing by the requirements, and the analysis of simulation results.

SREM is one part of a larger system, called the Distributed Computing Design System (DCDS), that covers the software development life cycle from the pre-software activities of system requirement definition to the detailed design of the distributed software. The other parts of DCDS are not treated in this paper.

2.2. A SREM Chronology

The activities and milestones pertinent to tracing the maturation of SREM into use on "real" projects are:

-- pre-1974: Research work on the formal definition of software requirements is carried out at various locations. In particular, work at the University of Michigan on the ISDOS system demonstrates the feasibility and desirability of using entity-attribute-relation concepts in capturing requirements in a database that can be processed to produce reports and help gain insight into the consistency and completeness of the requirements.

-- pre-1974: Work on automated support for the design, coding and testing of defense application systems is sponsored by BMDATC. In particular, a System Environment and Threat Simulator (SETS) is developed. It uses the stimulus-response definition of interactions among systems as a way of abstractly specifying the system so that simulation can be used to investigate the system's properties. Many of the concepts and procedures that later show up in SREM are manually used during this period.

-- late-1973 and early-1974: The interest at BMDATC expands to supporting the rigorous definition and validation of software requirements, in particular performance requirements. The SREM project is initiated.

-- August 1974: TRW, Huntsville, delivers a review of existing requirement definition and validation techniques and a plan for developing the SREM system.

-- October 1974: A specification of SREM and a set of requirements for RSL are finalized.

-- March 1975: A plan for testing* the SREM system is produced.

* A design for the REVS system does not appear for another seven months, so usage up until now, and the usage specified in this test plan, is presumably either manual or with the support of prototype tools.

-- July 1975: The concepts underlying SREM and RSL are presented at a conference on formal specification.

-- October 1975: A REVS design specification is delivered.

-- April 1976: RSL and the underlying graph model of computation are presented at a conference.

-- July -- September 1976: Manuals are provided for using REVS, maintaining REVS, and using the SREM method.

-- October 1976: The Software Development System, of which SREM is a part, debuts at the 2nd International Conference on Software Engineering. At this time, SREM is operational only on the Texas Instrument Advanced Scientific Computer (TI ASC) and one moderate-sized system has been attacked as a proof-of-concept demonstration of both the system and the method. SREM consists of 40,000 executable statements in Pascal and 10,000 lines of Fortran code re-used from the ISDOS system.

-- early 1977: SREM's initial release is available. SREM is ported to a TI ASC at another government installation.

-- June 1977: The results of experimental usage of the SREM system by TRW are delivered. This marks the end of SREM as a research project.

-- July 1977: Porting of SREM to BMDATC's CDC 7600 with on-line graphics capabilities is completed. Tuning of the system to take advantage of the machine's features results in a 20:1 increase in performance.

-- November 1977: A three-week training course is provided for practitioners from four companies and one government organization. Prior to this, training has been done, with only a modicum of success, by providing the documentation and, with considerably more success, by providing on-the-job-style instruction.

-- by November 1978: SREM has been transferred into other parts of industry with its successful porting to a CDC 7700 and a Cyber 74/174 TSS. By now it has been used on 13 "real" projects (eight of them within TRW) and the use has been to either validate existing requirements or to develop an initial set of requirements. The projects are all large scale (estimated code sizes for final products are 20-200K) and they span a variety of applications (operating systems, information management, man-machine interaction, data reduction, distributed processing, etc.), many of which were not in the class of problems originally envisioned for SREM.

-- by November 1980: SREM has been ported to only one more installation, an interactive Cyber 174/175 at a government organization. A new release has been prepared that has factors of 10 to 100 improvement in speed — what used to cost in the hundreds to thousands of dollars for computing resources is now costing in the tens of dollars.

-- by sometime in 1981: SREM has been ported to the VAX 11/780.

-- by October 1982: About 60 copies of the VAX version have been distributed.

-- during 1983: A SREM users' group is formed with about 20 organizations involved.

-- by late 1983: A two-year independent study of SREM has been completed to evaluate its applicability to C3I systems with an emphasis on how well it handles systems with man-machine interactions, large databases, and distributed networks of computers. Part of this involves encoding, in RSL, the requirements for a system that already has a 515-page English-language requirements document. This validation uncovers about 100 errors in the thought-to-be-correct requirements and requires about 5.6 person-years of effort over 20 months. Several problems are uncovered: SREM cannot easily handle parallel and distributed processing situations; poor syntax error handling and other problems create some usability problems; automatic invocation of some analysis capabilities leads to wastage of human and computer resources; and the training courses are considerably flawed. (The DCDS project is addressing these problems and later versions of SREM will have these problems corrected.)

-- by January 1984: About 80 copies of the VAX version of SREM have been distributed and about 20 additional requests for SREM are in processing. The majority have gone to industry, a few have gone to government installations, and one-third have gone to academic installations. 20-30 training courses based on the VAX version have been delivered. The database portion of the VAX version has been re-programmed in Pascal with a resulting improvement in the database portion of about 80:1. Other improvements have led to an overall improvement of about 16:1 over the initial VAX version. SREM is being used on at least six TRW projects both at Huntsville and at Redondo Beach; on some of these projects it was a management decision to use SREM even though its use was not required. SREM is also being used outside TRW.

2.3. <u>The SREM Technology Maturation Process</u>

Within this chronology we can find the roughly-defined historical periods depicted in Figure 1.

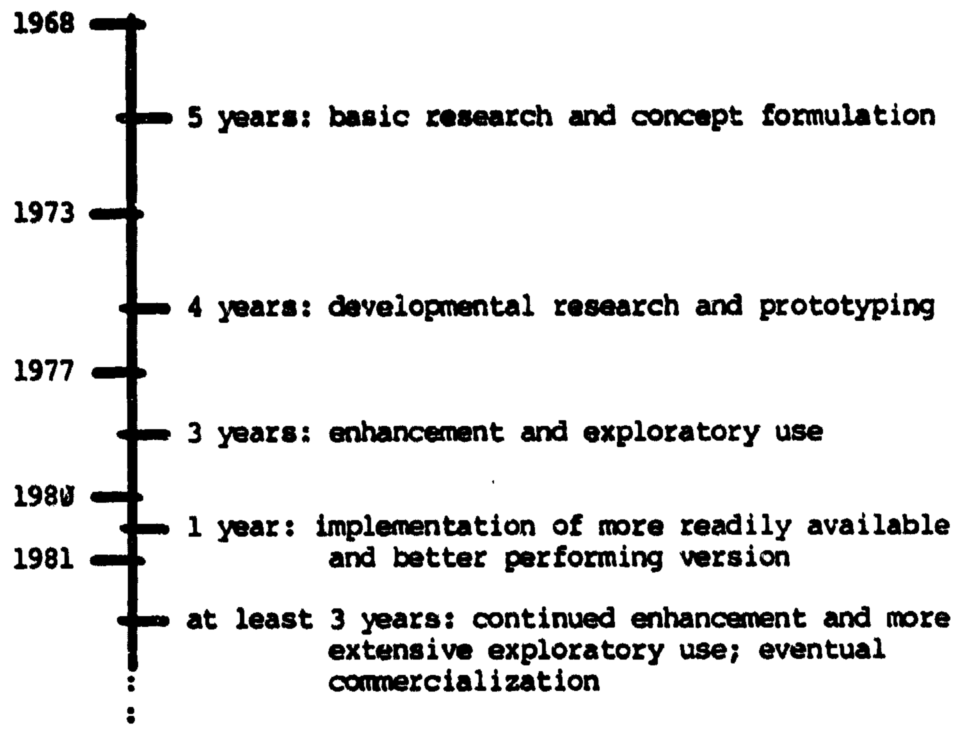

1968
5 years: basic research and concept formulation
1973
4 years: developmental research and prototyping
1977
3 years: enhancement and exploratory use
1980
1 year: implementation of more readily available and better performing version
1981
at least 3 years: continued enhancement and more extensive exploratory use; eventual commercialization

Figure 1: SREM Technology Maturation Time Line

As is true of most research situations, it is hard to identify exactly when the ideas started to emerge. Many of the concepts used in SREM were developed prior to the "official" start of the research in late 1973 and early 1974. The 1968 date is chosen since this is roughly when both the ISDOS and SETS projects started and these projects established some of SREM's underlying concepts.

The period up to 1977 is fairly typical, in content and duration, of a research project that has prototyping and proof-of-concept as its aim. The activity from 1977 on, however, seems a little slow. The early distribution of SREM was hampered by it being implemented on the TI ASC and the decision to port it to other super-level computers which typically have very individualistic operating systems which complicate the porting process. Had the decision, made roughly in 1980, to port SREM to a more widely available and obtainable computer been made two or three years earlier, the post-1977 chronology above would have been shortened, but probably not by more than a year.

SREM has not yet been commercialized. It is available, as are training courses and materials. And people are becoming attracted to it, at least to consider as an aid to developing the requirements for a system being built under government contract. But a fully commercial version is not available, and there is no active marketplace of sellers or buyers*.

* If our estimate is correct, then SREM will not reach this point until

3. The Unix Operating System

The Unix Operating System was developed at Bell Labs by Ken Thompson in 1969 in response to dissatisfaction with the Multics Operating System. It retained many of the basic concepts included in Multics but was implemented on a small mini-computer, the PDP-7. It was intended to be both highly supportive of the programming process and easily extended to include new capabilities.

In its 15-year history, Unix has become widely used both within and outside Bell Laboratories. The Unix community that has developed over the years includes not only users, but also operating system developers and researchers who find it valuable as a base for new operating systems and software engineering environments as well as an object of study itself.

3.1. A Unix Chronology

The activities and milestones pertinent to tracing the maturation of the Unix Operating System into widespread use are:

-- late 1969 and 1970: The initial version of the Unix Operating System is implemented on a PDP-7 mini-computer.

-- by 1971: The initial version is also operational on a PDP-9.

-- by 1973: Two more versions have been implemented, one for the PDP-11/20 and the other for the PDP-11/40 and PDP-11/45. The latter version has a size of 45,000 bytes. The user community for the latter version numbers 72 and this version exhibits very high reliability, being available about 98% of the time.

-- during 1973: Work is begun on the Programmer's Workbench version in an attempt to make the system useful for large projects carried out in a computing center environment. The previous versions have been more oriented towards small, cohesive groups of technology-oriented users and new facilities are planned for remote job entry, source code control, and testing on a variety of target systems.

-- summer 1973: The system is re-written in the C language. The size increases considerably (by about 33%) but the enhanced portability and maintainability are deemed worth it.

-- October 1973: The system debuts to the research community at the Symposium on Operating System Principles.

-- by 1974: Unix has been installed at 40 locations within Bell Laboratories. Its primary uses are: word processing, trouble data collection and processing, and order processing -- applications that are relatively far from the original intent of support for program development.

1986, plus or minus three years. It will be interesting to see if this holds true.

-- last half of 1970's: Considerable enhancement of the system is carried out at the University of California at Berkeley. This work leads to a stream of development that continues to be separate and parallel to the work done within Bell Laboratories.

-- mid-1978: A collection of papers appears as a special issue of the Bell System Technical Journal.

-- by 1979: Unix has been installed on over 2300 computers within Bell Labs as well as elsewhere in industry, Government and academia.

-- by 1980: Unix is available for all computers in the PDP-11 family, for the Vax 11/780, and for the Interdata 8/32.

-- by 1981: There are over 1700 Unix installations within colleges and universities. It is estimated that 90% of university computer science departments use Unix.

-- in 1981: A company, Unisoft, starts to provide Unix implementations for 32-bit micro-computers.

-- November 1981: Unix System III is announced along with a uniform pricing policy. Special licensing arrangements for added-value situations based on 32-bit micro-computers are also announced.

3.2. The Unix Technology Maturation Process

It is difficult to define historical "periods" with certainty, but the Unix chronology exhibits the rough pattern depicted in Figure 2.

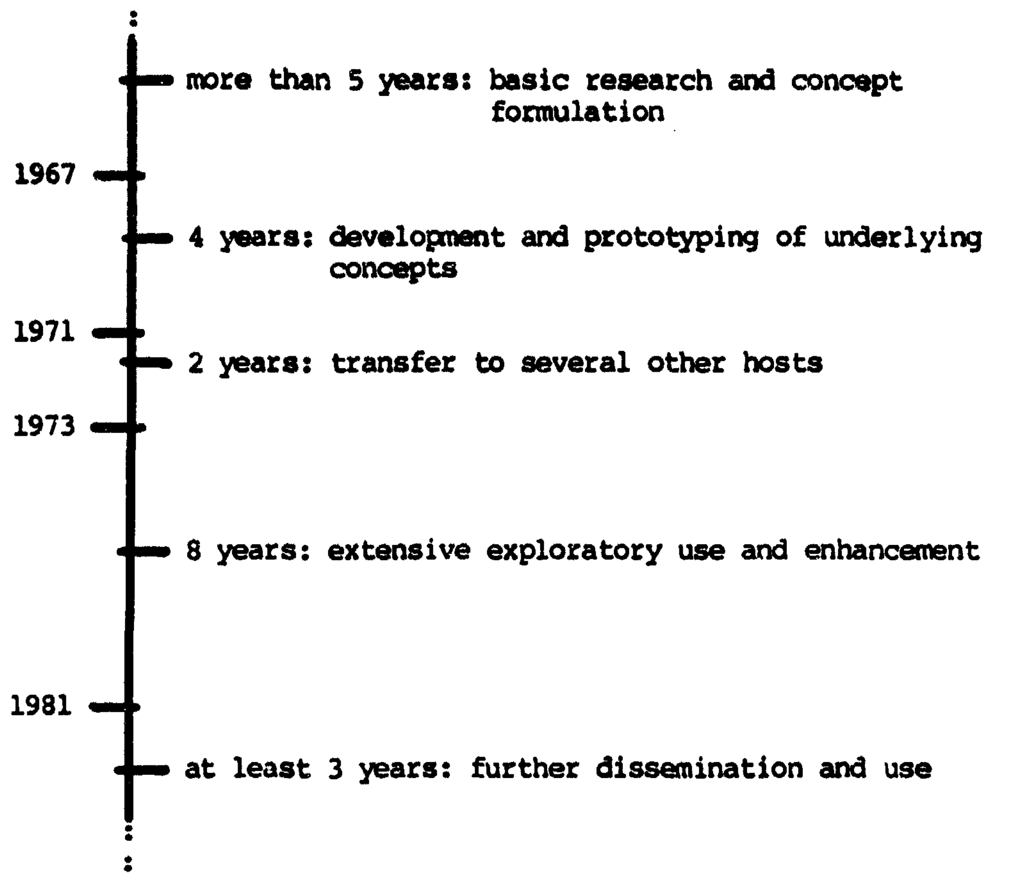

more than 5 years: basic research and concept formulation
1967
4 years: development and prototyping of underlying concepts
1971
2 years: transfer to several other hosts
1973
8 years: extensive exploratory use and enhancement
1981
at least 3 years: further dissemination and use

Figure 2: Unix Technology Maturation Time Line

Much of the heritage of Unix lies within the general operating system research of the 1960's and, in particular, the Multics Operating System. We can thus consider the operating system research of the mid 1960's as contributing to the development of the basic concepts underlying Unix. And, the Multics system can be considered to be an early prototype that helped in investigating the efficacy and compatibility of many of the concepts in Unix.

Unix enjoyed a relatively long period of prototypical use. It is difficult to say when it became an actual product and its use was more "serious" than prototypical. It was constantly under-going improvement and enhancement and so the achievement of a static point cannot be used in determining the end of prototypical use and the beginning of it being a real product.

Unix was available between 1973 and 1981 for a variety of machines and under a variety of licensing arrangements. The use during this period was heavily exploratory in nature, and Unix was still really in development as a commercial product. The appearance of Unix System III in 1981 marks the beginning of the treatment of Unix as a "product" by both the vendor and the marketplace.

Thus, the announcement of Unix System III is used as the dividing point between prototypical and "real" use. This is not because of the nature of this version or the realignment of licensing arrangements that took place. Rather, it is because this announcement coincides with the maturation of a rather extensive marketplace surrounding Unix. About this point in time training courses were becoming commonly available, texts on the system were appearing, service bureaus oriented towards Unix systems were established, alternatives to the Bell implementation were becoming common, and several companies were marketing systems hosted on Unix. Thus, the 1981 date marks the beginning of Unix as an element of some considerable strength in the marketplace as opposed to being just an available system having a dedicated following.

This makes the maturation period for Unix on the order of 20 years. Maybe this is a bit on the long side because of our decision to use the System III announcement as the commercialization point. But it is also true that the evolutionary development scenario that is one of the hall-marks of Unix tended to extend what we have characterized as the prototyp-ing and exploratory use phases. And it also seems true that the corporate decisions made within Bell and AT&T, during the period of divestiture, tended to slow down the commercialization of Unix.

4. The Smalltalk-80 System

The Smalltalk-80 System is the result of a number of inter-related projects within Xerox Palo Alto Research Park. The idea for a personal, electronic "notebook" emerged in the mid-sixties and was married with the concept of object-oriented programming at the beginning of the seventies. Other projects within Xerox have taken these concepts, individually and together, in other directions.

It seems fair to say that the world is fairly attracted to systems with bit-mapped displays and convenient interaction mechanisms like "mice". It also seems fair to say that the jury is still out on the concept of object-oriented programming. The former tends to heighten Smalltalk-80's commercial stance and the latter tends to detract from it. But, in sum, this system has made it to the fully commercialized point in its history.

4.1. A Smalltalk-80 Chronology

The activities and milestones pertinent to tracing the development and maturation of the Smalltalk-80 system are:

-- by 1968/1969: The concepts underlying a personal, pocket-sized infor-mation system are developed by Alan Kay in the course of his Masters and Doctorate work.

-- by October 1972: A first version of Smalltalk is prepared as a 1,000 line Basic program.

-- by late 1972: The first major version of Smalltalk, Smalltalk-72, is prepared by re-developing the Basic version in assembly code on a Data General Nova computer. This implementation is moved to the Xerox Alto computer as soon as possible. This initial version is used by about 12 people over the next four years.

-- by 1974: Several improvements have been made in the Smalltalk-72 interpreter to enhance the performance of the system.

-- by 1976: The second major version is prepared by making improvements to the Smalltalk language resulting in Smalltalk-76. This version is used by about 2Ø people on a daily basis and occasionally by about 1ØØ people.

-- by 1977: Much of the Smalltalk-76 system is ported to a suitcase-sized computer.

-- by 198Ø: The third major version is prepared by a further re-design of the Smalltalk language.

-- during 1981: The system is provided to four companies and one university who have agreed to try to port the system to other computers. The system has been split into a Virtual Machine and a Virtual Image that runs on the Virtual Machine. The companies and university have already received a specification of the Virtual Machine and have implemented it on various machines. Tapes of the Virtual Image are provided for them to get running on their Virtual Machines and a workshop is held to trade experiences in developing the Virtual Machines and porting the Virtual Image.

-- in mid-1983: Smalltalk-8Ø makes it to "commercialized" status with its announcement as an available product.

4.2. The Smalltalk-8Ø Technology Maturation Process

The historical periods in the lifetime of Smalltalk-8Ø are indicated in Figure 3.

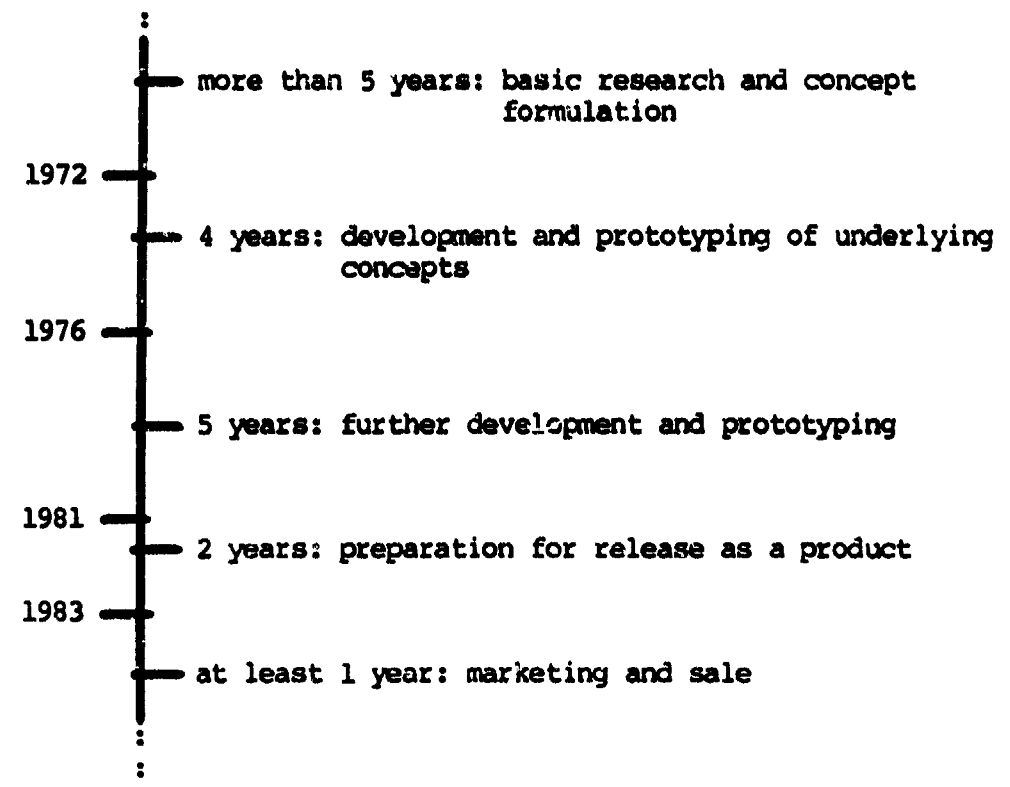

more than 5 years: basic research and concept formulation
1972
4 years: development and prototyping of underlying concepts
1976
5 years: further development and prototyping
1981
2 years: preparation for release as a product
1983
at least 1 year: marketing and sale

Figure 3: Smalltalk-80 Technology Maturation Time Line

The Smalltalk-80 system underwent a more organized and disciplined commercialization process than did Unix [10]. This process not only attended to getting the system transferred to other machines, but also served to get critical review from outsiders. Without a doubt, this was a much more efficient and effective procedure than the relatively haphazard enhancement and exploratory use phase that Unix went through. It has not only led to (marginally) faster maturation (roughly eighteen instead of twenty years) but also to a more technically coherent product.

5. Some Observations

The prototype version of even a part of a full life cycle software engineering environment can require massive computational resources. Large databases and friendly user interfaces can demand a base computing facility that is considerably more than one would expect the production version to run on. This is not to say that software engineering environment development plans that require a reasonable amount of computational resources cannot be devised. Rather, it indicates that careful, technically-accurate planning is needed to keep the computational resource requirements to a reasonable level.

The resources were kept to a minimum in the Unix and Smalltalk-80 initial implementations. In both cases, this requirement was a primary one for the project. But these systems were also intended, at the beginning, for use by an individual on small projects, and this helped considerably in keeping the resource needs low. SREM's orientation to use by a team of people working on a large-scale software system led to it starting with high computational resource needs.

Using large amounts of computational resources to get the prototype out quickly has to be traded off against the subsequent slowing of technology maturation due to complications in porting the system. In fact, it may be best not to waste time in porting the prototype and instead move directly to re-implementation (with as much re-use as possible). The SREM experience indicates some of the problems but does not decide the issue. The Unix and Smalltalk-80 approaches were to much more heavily rely on re-implementation.

Acceptance will depend on meeting some thresholds that are not known until transfer is attempted. For example, the cost* of requirements analysis with the initial version of SREM was roughly the same as the burdened cost of another analyst [6]. This would seem to be a reasonable incremental cost for the benefit obtained. But, SREM was not readily accepted until this cost was reduced by one or two orders of magnitude -- there seems to have been an unarticulated threshold concerning what the capability should cost and acceptance was delayed until this expectation was met. Unix and Smalltalk-80 did not seem to suffer problems in this regard, probably because they were heading toward a known domain and met the cost expectations held from experience rather than supposition.

Another milestone that must be met before acceptance of a system outside of its development community is use on a real project through delivery. This creates a real problem for a system, like SREM, that addresses the pre-implementation phases of large project -- there can be a considerable time lag (two to four years) before this milestone is met**. The situation is a bit more favorable for systems like Smalltalk-80 and Unix which address primarily the later parts of the software life cycle. But they can suffer time lag problems also. In government contracting, any tools to be used in the later parts of the life cycle must be specified before the project begins. That can mean a two- to four-year time lag between the decision to use a tool or system and its actual usage.

The time and resources required for distributing a system and responding to queries from potential and current users will be more than anyone would reasonably guess. The SREM experience was that at least as much time and resources were needed for distribution and "fire-fighting" as were needed for enhancement [6]. This "fact" is frequently missed in planning

---

* While this phenomenon is most easily discussed in terms of the dollar cost of using a capability, it is also related to the factors of response time and turn-around time.

** This time lag can be circumvented somewhat. For example, the SREM system can be used in requirements validation, rather than generation, mode out of a project's critical path.

out the scenario for system development and estimating the resources required and technology maturation can be considerably slowed as a result.

Under current government and industry practices and regulations, technology maturation will be hampered by rights issues. For example, SREM is government-owned and therefore in the public domain. The cost of making a production-quality version of SREM, to enhance its acceptability and therefore the extent to which it will permeate the software development community at large, will probably not be borne by private industry because of its public-domain status. TRW has funded the improvement of SREM's performance to be able to use SREM internally. Without government funding of the development of a production-quality version, a technology such as SREM will be an orphan without any visible means of support for the activities critically necessary for reasonable, let alone wide-spread, dissemination.

Innovative attempts to garner the resources for developing production-quality versions may be a way to temporarily improve the situation but these generally run into legal snags. The SREM users' group is one such attempt. And, while it did not have the intent of developing production-quality versions, the Smalltalk-80 "exercise" of transferring the system to other machines [10] contributed greatly to this end.

Extensibility of the system can greatly facilitate its transfer. For example, many of the transfers of SREM would not have been possible without the SREM capability to extend the RSL language while retaining the ability to analyze the new constructs with the existing tools.

Freedom to tailor the use of the automated tools is also important. Forcing users to employ the tools in a prescribed order is certainly important for enforcing a development method; but it can lead to a negative reaction even if the users are not trying to violate the method. Perhaps the users will want to use the individual tools in a different way without violating the method. Or perhaps they will find that enforced usage leads to unnecessary overhead when the method is used in their project with their management practices even though they are not violating the method. The latter situation was the case in the independent evaluation of SREM completed in late-1983. It was evidently generated by providing, in REVS, only a small number of large, monolithic tools rather than decomposing the tools into their constituent parts and allowing tailored use of the individual tool parts.

The independence of Unix from any particular development method was a definite contribution to its success -- since it could be rather freely molded to the practices of any user, it was more easily accepted. And the resulting large community of users meant that there was an extensive source of additional tools for importation and use.

This would not, however, have worked as quickly and extensively without the underlying "small is beautiful" philosophy of Unix. The Unix tools are really tool parts that one can easily combine through the available mechanisms of piping, job control programs, and tool embedding. As a result, new tools, that either provide a new function or deliver an existing function in a new way, can be easily and quickly prepared.

Another factor in the relatively quick acceptance of Unix was the presence of tool-building tools such as lex and yacc. These allowed new tools and tool parts to be easily and quickly developed and thus the set of available materiel could be easily extended and molded to new applications and new user communities.

As with most new technologies, acceptance of the sort of system talked about here is critically dependent on their meeting an already perceived need. The acceptance of Unix was greatly enhanced because it filled a recognized, felt need -- there was very little need to market it. SREM is considerably toward the other end of the spectrum -- only recently has the need for rigorous requirements definition and validation capabilities been fairly widely acclaimed. Smalltalk-80 seems somewhat in between -- open questions about the value of object-oriented programming will probably have a slowing affect on its acceptance.

In this regard, it is interesting to note the SREM experience that there is an increasing willingness to try new technology [6]. Partially, this comes from an increasing awareness of the value of pre-implementation tools such as offered by SREM. But there are several other factors: the use of advanced software development technology is starting to be required as part of some government contracts; the computing resources needed for using new technology is starting to be more commonly available, sometimes even at a practitioner's desk; and it is becoming more common for the associated acquisition and training costs to be moved into an organization's overhead expenses rather than charged against individual projects.

Finally, high-quality training is critical. For radically new technology, this training must accommodate the fact that the learning curve can initially halve the productivity of the technology's new users [5]. It must also be effective in spite of the initial hostility that the new users will have towards the new technology.

6. Acknowledgements

Sam Redwine first suggested to me that the period needed for the maturation of software support environment technology was more like 15 to 20 years than 10 years. His continued interest and his help in the development of this material was very beneficial.

Mack Alford, Chief Scientist at TRW Huntsville and father, midwife and mentor of the SREM system, was also very helpful. He provided me with many of the insights related in Section 3.

7. References

The study of SREM as an example of software technology maturation was facilitated by the availability of frequent periodic status reviews and evaluations. Much of the SREM literature of the 1970's is available in a edited collection of papers prepared by BMDATC [1]. This collection provides some early commentary on the SREM experience, including a 1978 review of the first two years of activities [2]. The next two years of activity are reviewed in [3] which includes a synopsis of some independent assessments and activities. Further information was provided as part of a 1982 program review [4]. An independent assessment of the maturity of SREM is available [5] as a result of a two-year study of SREM applicability to Command, Control, Communications and Intelligence (C3I) systems.

The literature relating Unix experiences is much more sparse. The original article on Unix [7] provides some status data and this is updated in a more recent paper [8].

The Smalltalk-80 experience is nicely captured in one of the recent books that have appeared on that system [9]. Of particular note, with respect to the issue of technology maturation, are the articles by Goldberg [10] and Ingalls [11].

1. C. G. Davis and C. R. Vick. Ten Years of Software Engineering. BMD Advanced Technology Center, Huntsville, Alabama. 1979.

2. M. W. Alford. The Software Requirements Engineering Methodology (SREM) at the Age of Two. Proc. Compsac78, Chicago, November 1978, pp. 332-339.

3. M. W. Alford. Software Requirements Engineering Methodology (SREM) at the Age of Four. Proc. Compsac80, Chicago, October 1980, pp. 866-874.

4. Distributed Computing Design System Quarterly Review. TRW, Huntsville, Alabama. October 1982.

5. P. A. Scheffer, A. H. Stone, and W. E. Rzepka. A Large System Evaluation of SREM. Proc. 7th Intern. Conf. on Software Engineering, Orlando, Florida, March 1984.

6. M. W. Alford. Private communication. 27 January 1984.

7. D. M. Ritchie and K. Thompson. The Unix Time-sharing System. Comm. ACM, 17:7, July 1974, 363-375.

8. R. W. Mitze. The Unix System as a Software Engineering Environment. In: Hunke (ed). Software Engineering Environments, North-Holland Pub. Co., Amsterdam, 1981.

9. G. Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley Pub. Co., Reading, Massachusetts, 1983.

10. A. Goldberg. The Smalltalk-80 Release Process. In: G. Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley Pub. Co., Reading, Massachusetts, 1983.

11. D. H. H. Ingalls. The Evolution of the Smalltalk-80 Virtual Machine. In: G. Krasner. Smalltalk-80: Bits of History, Words of Advice. Addison-Wesley Pub. Co., Reading, Massachusetts, 1983.

SDAM/15 April 1984

software design & analysis, inc.

1670 Bear Mountain Drive
Boulder, Colorado 80303

303 499 4782

Knowledge-based Systems as a Case Study in Software Technology Maturation

William E. Riddle

ABSTRACT:

Several knowledge-based systems are routinely used in problem solving tasks such as hardware system configuration and analysis of the chemical structure of hydrocarbons. In addition, some recent work work has demonstrated the utility of this technology as a basis for software environments. Thus, while this technology is not fully mature, it does provide an example of the process by which technology is developed from infancy to general use. The history of this technology is traced as an example of technology maturation.

This report was prepared as part of work performed for the Department of Defense STARS Joint Program Office by the Computer and Software Engineering Division of the Institute for Defense Analyses.

1. Introduction and Summary

Knowledge-based systems are software systems that provide intelligent problem-solving assistance that goes beyond the help for mechanistic activities that is typically provided. In particular, knowledge-based systems aid in the strategizing that occurs during problem solving.

The technology underlying knowledge-based systems has its heritage in artificial intelligence, emerging as a topic in its own right in the mid-1960's. The maturation of this technology is not yet complete but it has developed to the point that specific systems can be prepared in a relatively short time and several companies have started with the development of knowledge-based systems as the focus of their business.

In this paper, we trace the development of this technology to provide an example of the maturation of a general subarea within software technology. A brief introduction to knowledge-based systems and their use in support of software development and maintenance is given in the next two sections. Then the current status of knowledge-based technology is reviewed, followed by a history of this technology's maturation-related activities and a brief prognosis for the future.

2. Knowledge-based Systems

In a very real sense, all software systems are knowledge-based since they contain, in encoded form, knowledge about the problem being addressed, the solution being developed and the solution techniques being used. The term "knowledge-based" is used to distinguish those software systems that automate some aspect of the strategizing that occurs during problem solving. They are different from "information-based" systems that contain knowledge about the problem and its evolving solution, as well as the techniques useful in developing a full solution. In information-based systems, knowledge about <u>when</u> to do various activities and apply various techniques resides outside the system itself, within the users of the system. In knowledge-based systems, at least some of this strategy knowledge is held within the system itself and automatically applied.

Some of the strategy-related activities that might be (at least partially) automated in a knowledge-based system are:

-- determining which of several applicable techniques should be used,

-- assessing the applicability of various pieces of information and various solution techniques,

-- taking past experience into account,

-- learning new strategies (that is, ways to use the techniques),

-- innovating (that is, trying a technique even though one is not sure that it will work or developing new solution techniques),

-- realizing when limits of knowledge and capability are being reached and gracefully degrading in performance,

-- restructuring or abstracting the information base, and

-- explaining the logic underlying a solution.

To date, most knowledge-based systems have addressed only the first of these activities. In addition, the automated strategizing is most often interactive, involving the user in making the final decision after the system has narrowed the set of possibilities and derived information pertinent to the final decision.

For the most part, the development of knowledge-based systems faces the same problems facing the development of information-based systems. A conceptual basis for addressing a domain of problems must be developed. Appropriate problem-solving techniques and strategies must be found and made algorithmic. The techniques and strategies must be matured as a result of the experienced-based wisdom that evolves from their use. Ways of coping with limitations in capabilities must be determined. Techniques for providing insight into the behavior of a piece of software, particularly a malfunctioning one, must be developed. The limitations of traditional input/output interfaces must be surmounted. The software used in support of problem solving must be flexible, transportable and evolvable. And a wide variety of approaches must be captured so that individual differences in the user community can be accommodated. Of course, for developing knowledge-based systems all of these problems must be solved in a slightly different way -- namely, the solutions must have a higher degree of automatability.

There are at least three problems that seem to be unique to the development of knowledge-based systems. One of these is knowledge representation where the knowledge concerns the problem solving process. This involves not only the development of models of problem solving activities but also the preparation of problem and solution models that provide for a multiplicity of views at various levels of abstraction. Another problem that is relatively unique to the knowledge-based system arena is automated or computer-assisted knowledge acquisition, involving the drawing of inferences about problem-solving strategies from a record of a problem-solving activity and associated information. The third relatively unique problem is the development of approaches to handling "wrong" solutions -- for example, a good way to focus in on a solution is to consider incorrect solutions and determine why they are inadequate.

A good deal of literature has recently appeared on knowledge-based systems; [1] and [2] give particularly good treatments of this area and extensive references to both general literature and articles on specific topics or problems.

3. Knowledge-based Software Environments

The use of knowledge-based technology as a basis for environments supporting the development and maintenance of software systems is relatively new. [3] gives a good discussion of the prospects and focus of this approach to software environments.

Knowledge-based support for the software process (that is, the development and maintenance of software) can be understood by thinking of of the process as being carried out by a community of interacting agents, each having their own area of capability and expertise. One agent, for example, would be a compiler that receives information in the form of a source program and delivers new information in the form of an equivalent object program. Another agent would be a human who receives information in the form of test results and delivers new information in the form of error-correcting changes.

The intent of an automated software environment is to computerize the individual agents as much as possible. The intent of a knowledge-based automated software environment is to extend the scope of automation to those agents that perform feats of inference and logic that are well beyond what can currently be automated (for example, the inferences of changes to correct errors) or within the domain of strategic rather than mechanistic activities (for example, the choice of implementing storage structures). It is not the intent to totally automate the production and maintenance of software -- the critical need to keep the human "in the loop" at all levels of activity is well recognized.

4. Current Status

While the application of knowledge-based technology in support of the software process is still immature, progress has been made in several other application areas. Thus, we can get an idea of the maturation of this technology by considering it in general. In this section, we give a quick synopsis of the current state of the art and follow that, in the next section, with an accounting of some of the major events in the history of this technology.

A current-day knowledge-based system is typically composed of two major parts. First, there is the knowledge base which holds both the "passive" information about the problem and its evolving solution and the "active" information describing solution techniques and strategies. Second, there is the "user" of this information which can be thought of as a control program that uses the active knowledge to guide the transformation of the passive knowledge and move closer to a solution. Basically, the active knowledge provides "programs" for the various agents and the "user" segment of the system is an interpreter of these programs.

Most current knowledge-based systems encode the information about what a technique or strategy entails in the form of inference rules that specify actions to be taken whenever some specified condition holds. In many cases, these rules are encoded in the description of the "user" segment of the system -- otherwise, they appear as data in the active portion of the knowledge base. A severe problem at the current time, and a strong impediment to major progress, is understanding how to make the "user" a pure interpreter that, in and of itself, does not contain information about techniques and strategies.

For the most part, the rules are hand encoded and capture the accumulated experience and abilities of some human problem solver. In a few cases, the rules capture the experiences and abilities of a larger community of problem solvers but it is infrequent that the experiences and abilities of more than a hand full of "experts" are captured in a knowledge-based system. Current systems attend almost exclusively to the partial automation of the choice of techniques and few, if any, have any capabilities for carrying out the other problem-solving activities cited above.

The scope of current knowledge-based systems is generally fairly narrow. Most are very specific to particular applications but, none the less, contain parts that can be used in systems for other applications. A major goal is to localize problem domain information so that more general systems are available.

Knowledge-based systems are being actively and routinely used in several application areas. The most frequently cited areas are hydrocarbon structure analysis and computer hardware configuration. Other areas are geological exploration, speech recognition and lung disease diagnosis.

5. A Bit of History

Much of the heritage of knowledge-based systems is in the area of artificial intelligence (AI) and continuing development of this technology is integrally related to on-going work in AI. The chronology for knowledge-based technology is therefore almost identical to that of AI up until the early 1970's when the specifics of knowledge-based systems became a focus of attention in their own right.

The following gives a quick synopsis of this chronology. It comes primarily from historical reviews (perhaps the best being [4]) and from the personal experiences of the author who is not a researcher or worker in the knowledge-based area. While the dates for major steps are approximately correct, many more examples of perhaps better work could be cited.

prior to late-1960's: Artificial intelligence is a subject of active work from the very beginning of the computer era. Gradually over the years, the emphasis switches from doing intelligent tasks, like playing chess or checkers, to doing tasks intelligently, namely in much the same manner as a human would do them including the making of errors.

late-1960's: Early instances of "expert" systems appear, such as the Dendral system that aids in the analysis of the chemical structure of hydrocarbons. These provide intelligent assistance to human problem solvers and have the ability to solicit information that can be used in future problem-solving situations.

1973-74: Systems appear that separate the base of knowledge from the actions that work on and transform the knowledge, for example, the Hearsay speech recognition system at Carnegie-Mellon University.

1975: Articles start to appear on the "engineering" of knowledge-based systems.

1979: The effort needed to produce knowledge-based systems is down to about eight person years as opposed to more than 40 person years in the late 1960's. This can be partially attributed to an accumulation of ideas, code and experience. It can also be attributed to an increased understanding of the areas for which knowledge-based systems are most appropriate and most achievable.

1979-80: Several systems, for example Puff for lung disease diagnosis and R1 for hardware configuration, have matured to the point that they can be routinely used.

1981: The Japanese announce a major ten-year software technology program [2]. The major attention of the program is upon hardware to support inference computations and the software part of the program centers around the language Prolog. The program is therefore indirectly oriented toward knowledge-based systems.

1982: The Japanese indicate that the initial expectations for their program's results are probably too high but that they will gain a great deal of expertise and a number of side effects by striving for the announced goals of their Fifth Generation Computer project.

early-1980's: Companies are started in the United States to develop knowledge-based systems. These companies interview expert(s) in a proposed application area and construct a knowledge-based system for the application area. (Data are not available as to the success of these companies but they continue to thrive and do business.)

From this chronology, we can conclude that it took about eight years, from roughly 1965 to 1973, for the basic concepts underlying knowledge-based systems to emerge after the initial conception of these sorts of systems. Another six to seven years was needed to achieve usable capabilities. With the advent of the Japanese program and the appearance of companies based on knowledge-based system technology, the area has matured significantly but has not yet been widely extended in scope or usage.

6. Summary Remarks

Several systems based on knowledge-based technology are routinely used in problem solving tasks that are amenable to the inference-based approach that is currently most well-developed. And some impressive work has been done that demonstrates the utility of this technology as a basis for software environments [5].

But it will be quite a while before the technology, or environments based on it, are in common use within the software practitioner community. In fact, it appears that the best way to utilize the technology in the near future is to have "centers of excellence" where people experienced in and well-acquainted with knowledge-based technology can apply it and develop systems based on it. It is not clear whether or when the technology will become widely and commonly used within the software practitioner community.

REFERENCES

[1] A. Barr and E. Feigenbaum. The Handbook of Artificial Intelligence, Vol. 2. William Kaufman, Inc., Los Altos, California, 1982.

[2] E. Feigenbaum and P. McCorduck. The Fifth Generation. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[3] R. Balzer, T. Cheatham and C. Green. Software Technology in the 1990's: Using a New Paradigm. Computer, 16, 11 (November 1983), 39-45.

[4] Randell Davis. Expert Systems: Where are We? and Where Do We Go From Here? A.I. Memo No. 665, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1982.

[5] R. Balzer. Invited Talk and Demonstration at the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, April 1984.

SDAM/16 April 1984

software design & analysis, inc.

1670 Bear Mountain Drive
Boulder, Colorado 80303

303 499 4782

Abstract Data Types as a Case Study in Software Technology Maturation

William E. Riddle

ABSTRACT:

The history of the abstract data type concept is traced as an example of how basic, conceptual software technology matures and contributes to the maturation of other software technologies.

This report was prepared as part of work performed for the Department of Defense STARS Joint Program Office by the Computer and Software Engineering Division of the Institute for Defense Analyses.

1. Introduction

Abstract data types allow the natural and convenient separation of concern for an object's behavior from concern for its implementation. The concept emerged in the mid-1960's as part of the general "fountain" of ideas that are now fundamental to modern software technology. Since its emergence, the concept of abstract data types has matured to the point that it is commonly found in modern programming languages and has been critical to the development of other software technologies.

The basic concept of abstract data types is briefly explained in the next section in terms of the how it contributes to modern programming languages. Then a short history is given, followed by a short assessment of the current state of this technology and some predictions for its further propagation throughout the practitioner community.

2. Abstract Data Types

All programming languages provide a pre-defined set of data types and define a set of operations that can operate on these data types. For example, all scientific programming languages provide an array data type and at least the operation of selection of an array element by indexing and perhaps more sophisticated operations such as matrix algebra operations. An important point is that the user of the language does not know how the data types and associated operations are implemented -- the user's attention is focused on how the data type can be used rather than the details of how it is realized.

Abstract data types, as found in programming languages, are an extension of this idea to higher level data structures. Rather than predefine various higher level types, the user is provided with the ability to separately define two aspects of a type of data. First, the existence of the new type and the operations that can be done on it can be defined. Second, the representation used to implement instances of the type and the procedures used to implement the operations can be defined. Separate definition of these two aspects allows the use of instances of a high-level data type to depend only upon the type's external definition that spells out the operations that can be performed and how these operations behave. It also allows the implementation details to be encapsulated and hidden, providing the potential for being able to change them without disturbing the details of how they are used as long as the new representations and implementations continue to provide the behavior that is "advertised" by the external description.

3. A Bit of History

The ideas underlying abstract data types emerged in the mid-1960's and the technology has been developed to the point that abstract data types are part of most modern programming languages and have been used as a critical concept in other software technologies. Some of the time points in this history are:

-- mid-1960's: the concepts of classes and objects emerge as part of the Simula language

-- 1968: the concept of information hiding, key to the idea of abstract data types, is developed by Parnas and discussed in several technical reports

-- 1973: software development support systems appear that provide the ability to describe an object's behavior separately from the description of its implementation; at least one of these (TOPD) has its heritage in the Simula ideas of classes and objects; at least one other (HDM) is directly built upon the ideas developed by Parnas in his further work on information hiding; despite the variety of roots for these systems, they have essentially the same conceptual basis, a rudimentary form of abstract data types

-- mid-1970's: programming languages intended for the production of formally verified programs appear; these also contain preliminary versions of abstract data type facilities

-- 1977: Liskov and Zilles prepare a definitive article on the subject which is published in the premiere issue of IEEE Transactions on Software Engineering the following year; this paper, among other things, establishes a link between abstract data types and axiomatic specification

-- late-1970's: usable programming languages that include the notion of abstract data types appear; most notable among them is Ada

-- late-1970's: approaches to formal verification that rely on axiomatic specification are worked out

-- 1980: the Affirm system is operational; it relies on the concepts of axiomatic specification of abstract data types for much of it analysis of programs

4. Summary

The ideas of encapsulation and external description have been well received within the research and development community. They have been extensively developed and included in several other technologies such as programming languages and verification systems, much to the benefit of these other technologies.

Despite their extensive development and rather wide-spread application in other technologies, abstract data types are used by a relatively small proportion of the practitioner community. Primarily this is because of the lack of wide-spread use of modern programming languages that contain the concept of abstract data types. It appears that the further propagation of this technology is, at the moment, intimately linked to the more extensive use of modern programming languages.

Timeline for Development and Transfer of SCR Methodology

David Weiss

Naval Research Laboratory

February 1984

TIMELINE FOR DEVELOPMENT AND TRANSFER OF SCR METHODOLOGY

The following timeline shows key points in the development and transition of the Software Cost Reduction (SCR) project software development methodology. The reader should keep in mind that the SCR project is an attempt to refine and apply a combination of previously-suggested software development techniques. These techniques include information hiding, abstract interfaces, cooperating sequential processes, abstract data types, virtual machines, formal specification, program families, and undesired event (UE) specification. During the course of the project, some major new techniques have been developed and applied when no existing technique seemed suitable. Included are a new technique for software requirements specification and a new technique for specifying program semantics, along with its application to a new programming control structure.

To demonstrate the application of the SCR methodology, the operational flight program (OFP) for the Navy's A-7E aircraft is being redeveloped using the methodology. The OFP is a complex computer program that runs on the TC-2 computer on-board the aircraft.

The timeline is divided into two parts, the first representing events prior to the start of SCR that influenced the choice of techniques to be used by SCR. The second part represents events of interest subsequent to the start of SCR. Intervals on the timeline are measured in years. Points that appear prior to the start of the project represent one of the following:

1. The publication of papers in the technical literature describing or discussing new techniques later used in the SCR project.
2. The appearance of courses and tutorials discussing new techniques later used in the SCR project.
3. Completion of earlier demonstration projects showing the feasibility of new techniques without providing complete models. Points that appear after the start

of the project represent one of the following:

1. Completion of the development of a new technique or the refinement of an existing technique or combination of techniques. Included are model documents that provide an example of the application of the technique(s).
2. Publication in the technical literature of an exposition of the technique(s).
3. Completion of a subset of the OFP. Included is testing of the subset on the TC-2 simulator at the Naval Weapons Center. Such timeline points represent the completion of engineering models, and are of particular interest because the models include both tested code and the documentation used to develop and needed to maintain the code.
4. Completion of development of tools used in support of the SCR methodology.
5. Transfer of a subset of the SCR methodology, such as the requirements specification technique, to other projects.

The SCR methodology demonstration is considered completed when the engineering model corresponding to the entire A-7E (OFP) passes the same operational tests used to qualify existing A-7E OFPs for fleet service.

1968 First paper in technical literature on cooperating sequential processes.

1969

1970

1971

1972 First paper in technical literature on information hiding.

1973 1974 In-house working-papers describing UE handling.

1975 Early technical report on program families.

1976 First presentation of NRL software engineering course describing information hiding, program families, formal specification, UE specification and handling, abstract interfaces, and cooperating sequential processes.

Completion of in-house project demonstrating feasibility of program family and information hiding techniques in the development of a small simulator.

Timeline Part 1: Events Of Interest Prior To Start of SCR

1977 Start of Software Cost Reduction project - 6.1 phase.

1978 Requirements specification technique developed and model document published.

1979 Start of Software Cost Reduction project - 6.2 phase.

1980 Description of requirements specification technology published in technical journal. Abstract interface technique refinement completed and model document produced.

1981 Description of abstract interface technology presented at technical conference. Information hiding technique refinement completed and model document produced.

1982 Requirements specification technology in use by other projects and organizations*.

1983 First engineering model tested at NWC. Techniques used in the development of the model included information hiding, UE specification and handling, requirements specification, program families, virtual machines, and abstract interfaces.

Completion of initial support tools set. Tools included are those needed to support development and testing of the first engineering model.

Timeline Part 2: Events of Interest Subsequent To Start Of SCR

* Examples of other projects and organizations are: Bell Telephone Laboratories, NWC A-7 project, NWC/Grumman A-6E project, Air Force A-7 project, NUSC-Newport/SOFTECH Trident Emergency Preset System, Braun Bovieri.

SCR PUBLISHED PRODUCTS TO MARCH 1984

SOFTWARE REQUIREMENTS SPECIFICATION

Heninger, K., Kallander, J., Parnas, D., and Shore, J.; Software Requirements for the A-7E Aircraft; NRL Memorandum Report 3876; 27 November 1978. Release 5 in preparation.

Heninger, K. L.; "Specifying Software Requirements for Complex Systems: New Techniques and their Application", IEEE Trans. Software Engineering, vol. SE-6, pp. 2-13, Jan. 1980.

MODULARIZATION OF A COMPLEX SOFTWARE SYSTEM

Britton, K. H., and Parnas, D. L.; A-7E Software Module Guide, NRL Memorandum Report 4702, December 1981. Release 2 in preparation. Clements, Parnas, Weiss; "Enhancing Reusability with Information Hiding", Proceedings, Workshop on Reusability in Programming, sponsored by ITT Programming, 7-9 September 1983.

Clements, Parnas, Weiss; "The Modular Structure of Complex Systems," Seventh International Conference on Software Engineering, March, 1984

ABSTRACT INTERFACE DESIGN

Britton, K. H., Parker, R. A., and Parnas, D. L.; "A Procedure for Designing Abstract Interfaces for Device Interface Modules", Proceedings, Fifth International Conference on Software Engineering, 1981.

Clements, Faulk, Parnas; Interface Specifications for the SCR (A-7E) Application Data Types Module; NRL Report 8734; 23 August 1983.

Parker, A., Heninger, K., Parnas, D., and Shore, J.; Abstract Interface Specifications for the A-7E Device Interface Module; NRL Memorandum Report 4385; 20 November 1980. Release 4 in preparation.

Britton, Clements, Parnas, Weiss; Interface Specifications for the A-7E (SCR) Extended Computer Module; NRL Memorandum Report 4843, Jan 1983. Release 6 released 4 December 1983.

Clements, P.C.; Function Specifications for the A-7E Function

Driver Module; NRL Memorandum Report 4658, October 1981.

Clements, P.C.; Interface Specifications for the A-7E Shared Services Module; NRL Memorandum Report 4863, 8 September 1982.

REAL-TIME PROCESS SYNCHRONIZATION

Faulk, Parnas; "On the Uses of Synchronization in Hard Real-Time Systems", Proceedings, 1983 IEEE Real-Time Systems Symposium, 6-8 December, 1983.

A NEW CONTROL CONSTRUCT FOR AVIONICS COMPUTER SOFTWARE

Parnas, D. L.; An Alternative Control Structure and Its Formal Definition Technical Report FSD-81-0012, Federal Systems Division, IBM Corporation, Bethesda, MD., 1981.

BASELINED BUT UNPUBLISHED SCR PRODUCTS

ABSTRACT INTERFACE DESIGN

Campbell, Clements, Labaw, Parker; Interface Specifications for the SCR (A-7E) Physical Models Module; NRL report to be published; design of Earth Characteristics submodule has been completed.

Clements; Interface Specifications for the SCR (A-7E) Data Banker Module; NRL report to be published; baselined draft 15 September 1983.

HOW TO SPECIFY AN ABSTRACT INTERFACE

Britton, Clements, Parker, Parnas, Shore; A Standard Organization for Specifying Abstract Interfaces; NRL Report in preparation for publication; baselined draft 15 November 1983.

Clements, Parnas; "Experience with a Standard Organization for Specifying Abstract Interfaces," accepted for presentation at a jointly-sponsored conference on specification languages and techniques, March 1984.

SUPPORTING TOOL SET

Alspaugh, Clements, Mullen, Parnas, Weiss; Interface Specifications for the SCR Translator Tool Set, NRL Report to be published, baselined draft 15 September 1983.

Clements; Interface Specifications for the SCR (A-7E) System Generation Module; NRL report to be published; baselined draft January 1983.

IMPLEMENTATION DESIGN LANGUAGE

Faulk, S. R.; Pseudo-Code Language for the A-7E OFP; NRL Technical Memorandum; April 1982. To be published as an NRL Memorandum Report.

A NEW CONTROL CONSTRUCT FOR AVIONICS COMPUTER SOFTWARE

Parnas, D. L.; Less Restricted Structured Program Constructs, NRL Report to be published, draft June 1983.

TEST PROCEDURES

Clements, Parnas, "SCR Testing Guidelines", Technical Memorandum, draft 2 October 1983.

DISTRIBUTION LIST FOR PAPER P-1788

CPT David Boslaugh
2221 Jefferson Davis Highway, Rm. #944
Arlington, VA 22202

Paul Clements
Naval Research Laboratory
Code 7595
Computer Science and Systems Branch
Washington, DC 20375

Charles Colello
Plans and Programs Division
Rm 1D679, Pentagon
Washington, DC 20310

LTC Harrington
HQ AFLC/MMEC
Wright Patterson AFB, Ohio 45433

Jim Hess
DARCOM
9N23 AMC
5001 Eisenhower Ave.
Alexandria, VA 22333

John Leary
STARS Joint Program Office
400 Army Navy Drive, 9th Floor
Arlington, VA 22202

Edward Lieblein
OUSDRE/R&AT (CSS)
400 Army Navy Dr., 9th Floor
Arlington, VA 22202

LTC Vance Mall
OUSDRE/CSS
400 Army Navy Drive, 9th Floor
Arlington, VA 22202

Robert F. Mathis (25 copies)
Director, AJPO
400 Army Navy Drive, 9th Floor
Arlington, VA 22202

Carol Morgan
400 Army Navy Drive, 9th Floor
Arlington, VA 22202

LTC Mote
HQ Air Force Systems Command
Office Code ALR
Bldg. 1535 Rm. EE205
Andrews AFB, MD 20334

COL Ken Nidiffer
HQ Air Force Systems Command
Office Code ALR
Bldg. 1535 Rm. EE205
Andrews AFB, MD 20334

Jim Riley
HQ AFSC/DLA
Andrews AFB, MD 20334

Dick Stanley
STARS Joint Program Office
400 Army Navy Drive, 9th Floor
Arlington, VA 22202

Hank Stuebing
Code 50C
NAVAIR DEVCEN
Warminster, PA 18974

David Weiss
Naval Research Lab
Code 7592
Computer Science and Systems Branch
Washington, DC 20375

Other

Betsey Bailey
400 N. Cherry Street
Falls Church, VA 22046

John Bailey
400 N. Cherry Street
Falls Church, VA 22046

Barry Boehm
TRW Defense Systems Group
MS R2-1076
One Space Park
Redondo Beach, CA 90278

Bill Carlson
Intermetrics
4733 Bethesda Avenue, Suite 415
Bethesda, MD 20814

Ruth Davis
The Pymatuning Group, Inc.
2000 L St., N.W., Suite 702
Washington, DC 20036

Richard DeMillo
Georgia Institute of Technology
School of Inf. and Computer Science
Atlanta, GA 30332

Larry E. Druffel
Rational Machines
1501 Salado Drive
Mountain View, CA 94043

Frank McGarry
NASA/GSFC
Code 582
Greenbelt, MD 20771

John Manley
Computing Technology Transition, Inc.
82 Concord Drive
Madison, CT 06443

Ann Marmor-Squires
TRW
Software Development Lab
2751 Prosperity Ave.
Fairfax, VA 22031

Ronnie J. Martin
School of Information & Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Don Philpot
Software Engineering Technology Corporation
197 Montgomery Rd. MC3
Altamonte Springs, FL 32714

Defense Technical Information Center - (12 copies)
Cameron Station
Alexandria, VA 22314

William Riddle
Software Design and Analysis
1670 Bear Mountain Dr.
Boulder, CO 80303

DoD-IDA Management Office
1801 N. Beauregard St.
Alexandria, VA 22311

<u>IDA</u>

Ms. Louise Becker
Mr. Matthew Berler
Mr. J. Frank Campbell
Dr. Jack Kramer
Ms. Sarah Nash
Dr. Thomas H. Probert
Mr. Samuel T. Redwine, Jr.
Mr. John Salasin
Dr. Marko M. Slusarczuk
Mr. E. Ronald Weiner
Ms. Carol Powell - (2 copies)